Original

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE 12/9/2005 (MM/DD/YY)	3. REPORT TYPE AND DATES COVERED Final Report, April 11, 2005 to December 11, 2005

4. TITLE AND SUBTITLE
Agent Based Computing Machine

5. FUNDING NUMBERS
FA8750-05-C-0127 CLIN 0005

6. AUTHOR(S)
Edwin R. Addison

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Lexxle, Inc.
1121 Pembroke Jones Dr., Suite 200
Wilmingotn NC 28405

8. PERFORMING ORGANIZAT REPORT NUMBER
FA8750-0005-2

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFRL

10. SPONSORING/MONITORI AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
NONE

12a. DISTRIBUTION/AVAILABILITY STATEMENT
[✓ Approved for public release; distribution unlimited.]
~~Distribution authorized to U.S. Government Agencies only; contains proprietary information.~~

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)
This SBIR developed a new class of computer architecture called an "agent based computing" module (ABC Machine) that enables "cognitive computing" algorithms to be implemented effectively on a large scale. The ABC Machine is a biologically inspired architecture derived from the field of "membrane computing" and is also based upon "statistical dataflow computing". It operates in local contexts over string operators. The ABC Machine is motivated by analyzing the biochemical processing in cells. The architecture is suited for computing problems not easily solved by traditional machines. It has the properties of very high parallelism, distributed and redundant processing, and graceful degradation. Phase 1 demonstrated advantages over traditional AI algorithms on conventional machines in the following ways: 1) pattern recognition problems with very large numbers of

14. SUBJECT TERMS
SBIR Report, cogntive computing, biologically inspired computing, artificial intelligence, agent based computing, pattern recognition, complex systems

15. NUMBER OF PAGES
49

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRAC

Agent Based Computing Machine

AFRL SBIR Phase 1 Final Report
Lexxle, Inc.
FA8750—05-C-0127

Table of Contents

1. Identification and Significance of the Problem or Opportunity.

This section introduces the concept, sample algorithms, architecture and proposed implementation of a biologically inspired computing system called an "agent based computing" machine (ABC Machine). The ABC Machine is motivated by cellular biochemistry and it is based upon a concept called "statistical dataflow computing" and it operates in local contexts over string operators. It is modeled in part by analyzing the biochemical processing in biological cells, but with a goal to provide an architecture for computing problems not easily solved by traditional machines such as pattern recognition, predictive modeling and logical reasoning. The ABC Machine is related to a recent biologically inspired computing concept called a "membrane computer" (Calude and Paun, 2001). However, it has been extended to enable string processing and large scale redundancy to more easily accommodate cognitive computing algorithms. While the membrane computer is an abstract computing concept that has never been built, the ABC Machine is a special case of a modified membrane computer with a practical architecture. It has been shown in Phase 1 to be effective on cognitive computing and/or artificial intelligence algorithms.

The ABC Machine can be characterized as a statistical dataflow machine that computes over string operators using massively parallel redundant rules in contained localized computing regions (called "membranes"). Computation is mediated by specific object classes that regulate how much and which instructions can execute (these object classes serve roles similar to energy molecules and enzymes in cells), thus enabling context sensitive computing. The architecture of the ABC Machine is explained in greater detail below. The ABC Machine "computes" by statistical approximation using redundant rules in virtual computing regions (membranes). The ABC Machine processing is context dependent, allows I/O operations across these regions or "membranes" and can be programmed manually or by genetic programming. The regions act as "agents" and memory is distributed throughout and across regions. It offers the promise of a machine that can solve complex pattern recognition problems, complex simulation problems and logic problems such as expert systems or production systems. Cognitive computing problems may be modeled by ABC Machine computation using a series of statistically redundant strong objects and context sensitive operators.

A fundamental difference between an ABC Machine and a conventional von Neumann computer is that in the ABC Machine, computation is fuzzy and redundant. The results of any computation are based upon the "state" of the machine and are often determined by a voting or majority rules concept for the redundant fuzzy computations. This is what is meant by "statistical" computing. A second fundamental difference is that it is a dataflow machine, meaning that instructions only execute when their operands "arrive" for computing as opposed to the traditional fetch cycle in a control flow architecture.

The remainder of this section describes the ABC Machine by first giving a conceptual overview to illustrate the processing concepts. This is followed by the presentation of several sample algorithms and algorithm classes and how they work. The special area of cognitive computing is addressed. The architecture is then formally defined, followed by a brief computational analysis. Several options for hardware implementation are explored including an emulation with a high performance cluster, a high performance silicon chip and the potential for biochemical implementations.

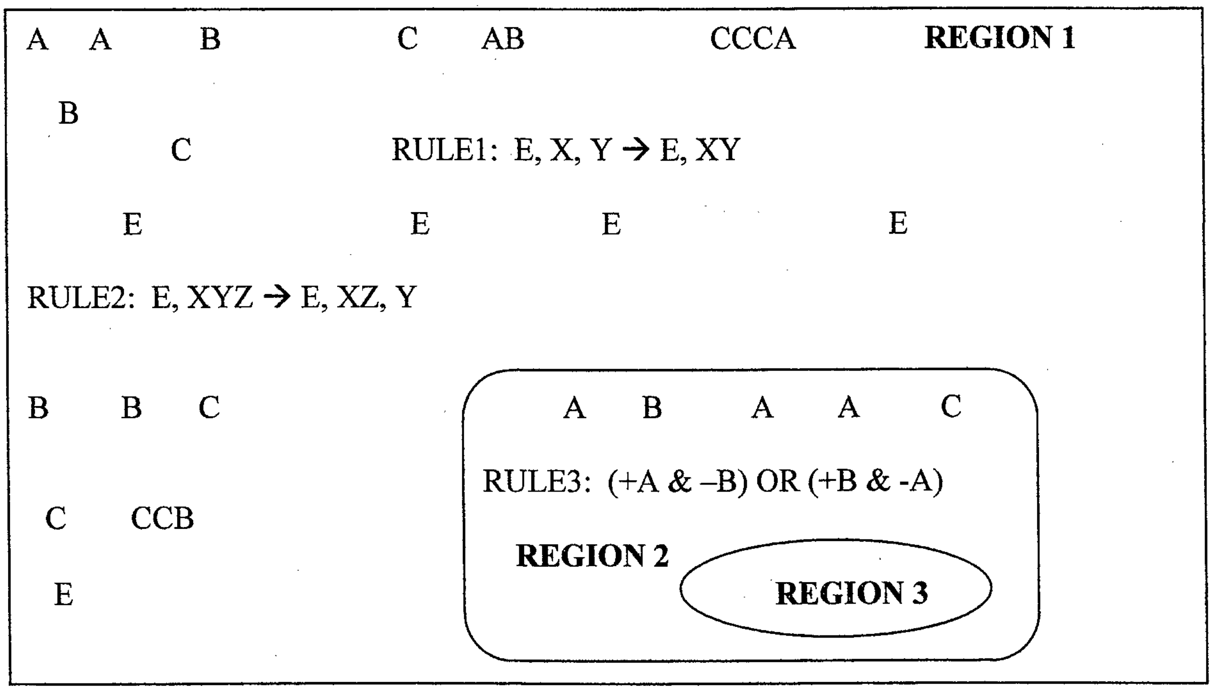
1.1 Conceptual Overview of the ABC Machine

The ABC Machine is a work in progress on US Department of Defense Small Business Innovation research funding. While its physical implementation could take various forms ranging from a single CPU emulation to a custom board to a biochemical solution, this proposal focuses on a high performance cluster (HPC) implementation. The ABC Machine is a distributed, parallel architecture that operates over large redundant rule and data sets. It is a "statistical dataflow machine" that operates in a nondeterministic manner. It is a derivative of the concept of membrane computers or "P Systems" and is inspired by biological principles. It augments a classical P System in that it operates over string operators, has higher level system functions, has a distributed content addressable memory (CAM), various predefined string and communication operators, and the potential to implement local regions with specific system functionality. It can also be thought of as a super generalization of a neural network. The nature of the ABC Machine renders it highly useful for cognitive computing tasks. This concept is explored in further detail in the remainder of this section and those that follow.

To illustrate the concept of how ABC Machine computation works, consider the specific example of an ABC Machine computing node illustrated in Figure 1 below. An ABC Machine consists of many such nodes (implemented in software or hardware). The node contains tokens (such as represented by symbols A, B and C) and "strings" of tokens (such as CCCA and CCB) and rules (RULE1 and RULE2). This diagram represents an "ABC Machine" at an instant in time. The regions are nested and each is a particular computing zone isolated from the other regions. In this case, Region 3 is empty. The alphabetic symbols (A, B, AB, CCB, etc.) are "strings" of atomic elements. The ABC Machine computes over strings and modifies them using the rules. There may be many copies of any given string and/or rule. Rules execute when operand strings become available.

It also contains a nested node or "membrane" (Region 2) that has tokens and an I/O Rule (RULE3) that allows the exchange of A tokens for B tokens or visa versa with the Region 1 membrane. In Region 1, RULE1 concatenates two tokens or strings and RULE2 removes the middle out of a string that is at least three elements long. The rules execute when the object "E" (called an 'enzyme' for its analogy to biology) is present and when the respective operands are within the region of the rule. Depending on the physical implementation of the computer, random selection can be used among all those rules that qualify per the previous statement. So a plausible sequence of events is eventually Region 2 exchanges the entire A elements for B elements in Region 1 and then RULE1 eventually builds up a string AAAAA. This event might be deemed improbable if all objects move randomly and therefore many other possibilities exist.

Figure 1. Illustration of an ABC Machine Computing Node



The illustration above shows only single copies of instructions or rules. The ABC Machine, however, is defined to be a statistical machine, meaning that there are many copies of each instruction and operand, in fact very large numbers of each. They operate redundantly and in parallel. Computational results are based on the quantity of a particular object being produced. So unlike traditional computing, an ABC Machine is intended to be parallel and redundant with very large numbers of copies of all objects and instructions. The execution of which occurs first or when is random. So unlike a von Neumann machine where a poorly executed instruction is a catastrophic error, here it simply becomes one bad computation that will be buried by many good ones – hence providing a graceful degradation. This concept is motivated form the principles if biology where everything occurs in large numbers and results are based upon the concentration of a particular biochemical rather than a single molecule.

At any instant in time, the "state" of the ABC Machine is the set of strings that exist in each region. The ABC Machine "computes" by applying rules in a region to the strings in a region in an asynchronous, maximally parallel manner. A problem begins with an "initial state", that is the set of regions, the set and arrangement of strings in the regions, and the operators or rules that exist in each region. Over time, the state evolves due to the execution of the rules, including the movement of strings across region boundaries when rules call for this. The solution to a problem is represented by the final state after computation is finished, or in some cases, the state "trajectory", that is, the evolutionary path of state. Usually, the solution to a problem will be contained in a small portion of that final state.

The ABC Machine is actually a special case of an abstract machine class reported in the literature called a membrane computer (Calude and Paun, 2001).). A membrane computer can be represented by a Venn diagram with no intersecting regions similar to Figure 1. Each region is an area for computation. It is motivated by the biochemical functioning of biological cells. Outputs of computation in one region make inputs to other regions. Calude and Paun have addressed membrane machines in theoretical depth. They describe a system where "objects" inside membrane regions "evolve" through a series of computational steps and rules. The theory of membrane computing is reasonably well developed, although no membrane computers have been built.

The ABC Machine is, in part, motivated by the theory of membrane computing, but it is also driven by the need to solve practical problems. Because both the membrane computer and the ABC Machine operate on symbols rather than numbers, it potentially makes an ideal candidate for cognitive computing. That is the main thesis of this proposal and the main goal of the Phase 1 effort is to prove and demonstrate that.

The ABC Machine is a membrane computer, but it exhibits more general computing features than those reported in the membrane computing literature. The following specific features are examples of computing functions that appear in Figure 1, but are not reported on in the membrane computing literature in any depth:

- Use of strings as objects instead of atomic objects
- Use of multiple copies of operands and multiple copies of instructions operating redundantly
- Using the notion of distance or proximal location as a prerequisite for rule execution
- Implementation of I/O rules
- Illustration of rules which require enzymes (the object E) to be present in order to execute.

The illustration above showed a "ligation" rule (RULE1) that merges two objects together to make a longer string (i.e. X and Y become XY), a "digestion" rule (RULE2) that takes a string and breaks it down into two smaller objects, and an I/O rule (RULE3) that exchanges objects between two regions. Digestion rules are useful in computational work involved in sorting, matching or logical comparisons. Ligation is the opposite of digestion and it is the process of concatenating two strings. Once digestion and ligation are defined, logic operations can be performed. For example, a NAND gate is a function that means "not AND". In traditional computers, it returns a value of 0 when all of its inputs are 1, and otherwise it returns a 1. Things are not so simple in the ABC Machine world, as there is no absolute 1 or 0 answer to a question, but rather a statistical concentration indicating an expression level. Tokens and strings are not variables, they are absolute symbols and cannot take on a 0 or 1 value. They are absent or present.

An AND gate is a simple ligation instruction of two or more inputs in a specified order. The output is the ligated string and the answer to the AND function is the concentration level or expression level of the output product. This, in shorthand notation, A AND B can be implemented by the following ligation: (A, B) → AB. In other words the string A is appended to the string B to make AB.

The "expression level" of AB, or <AB> is the result. The expression level means "the quantity of", so <AB> represents a number corresponding to how many copies of object AB are present. A high value of

<AB> means A AND B is true. A NAND gate requires no further implementation, as a low value of <AB> indicates A NAND B is true.

Statistical computing output is always in expression level rather than absolute values. The very meaning of a logic gate operator is changes. In a von Neumann architecture, the gates AND and NAND have absolute meaning. In an ABC Machine architecture, they have only expression levels. The tokens and/or strings with high expression levels represent the state or answer to a problem. This is not unlike gene expression or biochemical reactions in cells. There is only an expression state and never an absolute state. This will be significant later when programs and algorithms are discussed.

So, an OR gate can be computed from a ligation instruction also, but other operands must be involved. For example if it is desired to compute the value of A OR B, this can be done by ligating A and B separately with other commonly available operands and then digesting the A or the B off the resulting string. This would proceed as follows. The ligation operations (in shorthand notation) are:

(A, X, Y) → AXY
(B, X, Y) → BXY

Here it is assumed that X and Y are highly expressed in the region of computation. This ligation instruction is followed by a digestion instruction, as follows:

AXY → (A, XY)
BXY → (B, XY)

The value of A OR B can be ascertained by measuring the expression level of XY as <XY>. If <XY> is high, then A OR B is considered to be true. Once again, recall that A and B (and X and Y) are tokens or strings and not algebraic variables. There are no algebraic variables in instructions. The data and programs are inseparable as in LISP programming in the conventional computing world. Logic gates in an ABC Machine must be interpreted statistically. The execution of any one copy of the logic gate instruction is insignificant. What is significant is the concentration of operands and products indicating in a large sense the relative quantity of execution of the logic gate instruction. Thus, the "answer" is not black or white, but gray, but perhaps the gray points toward a black or white answer.

1.2 Biological Motivation

DARPA's current BICA program (Biologically Inspired Computer Architectures) (BICA), is evidence of current interest in the notion of using biology as a metaphor for novel computer architectures. Before identifying the architecture and components of the ABC Machine, it is illustrative to identify the computational components of a biological cell and make a functional analogy to traditional computing architecture. Several authors have addressed this idea, but a full description is not yet realized. Cray (1996, May 30) made a correspondence between various operating system components and the functioning parts of a cell.

A cell may be viewed as a computing element or processor with the following characteristics. It is an interrupt driven dataflow machine that processes many instructions in parallel and uses many subordinated processors. Ultimately, the cells role is to send and receive various communications (electrical, chemical) with other cells at various points in time. The purpose of its internal "computation" is to stay alive, recycle "operands", express selected genes (i.e. run programs), and produce output. The output produced by the cell can be viewed as information, although it often takes a physical form (molecules). The cell phenotype is a physical response to gene expression (i.e., programs that have run). It can be viewed as a "state vector".

Unlike a traditional computer that purely produces abstract information, the cell often produces physical information and it changes in structure and state as it does so. Therefore, a cell can be viewed as more than an information processing engine – it is also a molecular machine. Therefore, a cell may be viewed as a computer of sorts (Cray ,1998), (Paton, 1994). The following table makes a partial comparison of the components of a computer with the functioning elements of a cell that extends the ideas of Cray (1998) and Paton (1994). Key features of a computer are matched with a related feature of a cell that accomplishes functionally similar things.

The computer program of a cell is its DNA. The DNA maintains copies of nucleotide sequences that encode for proteins, which perform functions in cells. A given gene (a subset of the DNA) is transcribed into mRNA when appropriate transcription factors (specialized proteins) are present. Hence, mRNA acts as a cache memory or copy of a specific instruction in a temporary location. When a gene is transcribed, it is said to be expressed, and its expression level is the number of copies of mRNA currently present in the cell and thus can be thought of as the "state" of cell with respect to a specific gene at a given point in time. The cell has an operating "kernel" which translates mRNA into proteins. There are some notable differences between the functioning of a cell and that of a von Neumann machine. A few of the most important differences are listed below:

1) A cell is a dataflow machine. Instructions (i.e. chemical reactions) execute when their operands (i.e. reactants and enzymes) arrive on the scene. There is no sequential program of instructions as in the von Neumann machine.

2) Instructions operate in a statistical manner, by having a large number of redundant copies of each instruction and operand. Therefore, the chemical reactions in a cell are significant only when they occur in large concentrations exceeding a significance threshold. The behavior of such concentrations of proteins is driven by the law of large numbers. The well known chemical processes of diffusion, active transport, enzyme driven chemical reactions, and metabolism drive the production, location and concentration of cellular reactants (Becker, 2000). Unlike the von Neumann machine, a single instruction execution by itself is not significant unless it occurs a significant number of times producing a concentration of products (operands for instructions).

3) A series of instructions or reactions in a cell forms a pathway or circuit representing a sequence of instructions as in a subroutine or function call in a von Neumann machine. What is different, however, is that such pathways execute in large numbers and in parallel with many other active pathways at the same time. Thus, the cell is not only a statistical machine, but it operates as a parallel, distributed machine.

4) The cell may be thought of as a processor or CPU. In a von Neumann machine, there is one (and sometimes a few) CPUs. In a cellular environment, there must be many cells in order for useful processing to occur. For example, a tissue or organ consists of a very large number of cells acting cooperatively. Hence, the ultimate deployment of a cell is in a large "swarm" of other cells in a network, rather than a single CPU integrated with peripherals and a control system. Cellular behavior in large numbers or in a network is driven by competition for energy and survival (Bar-Yam, 1993)

5) In a cell instructions are hardware elements (proteins) or actions of hardware elements (chemical reactions). The reactants and products can be digested, synthesized, ingested, or exported. They are tokens or objects. Hence, the operating environment of a cell is highly object oriented.

The computer architecture most similar to cells is "Membrane Computing" (Paun, 2001). A membrane computer looks to the whole cell structure and functioning as a computing device. The membranes within a cell play a fundamental role as filters and separators. A Venn diagram with no intersecting regions can represent a membrane computer. Each region is an area for computation. Outputs of computation in one region make inputs to other regions. Calude and Paun have addressed membrane machines in theoretical depth. He describes a system where "objects" inside membrane regions "evolve" through a series of computational steps and rules. A computational system, called a "P System" will halt when no object can further evolve. Calude and Paun (2001) explore several variants of P Systems from a theoretical perspective by investigating properties such as decidability conditions and normal forms.

A P System (membrane computer) is a distributed and highly parallel model based on the notion of a membrane structure. The structure consists of cell-like membranes recurrently placed inside a "skin membrane" (Calude and Paun, 2001). As stated above, it can be represented as a Venn diagram with nonintersecting regions. Each region may contain objects that can evolve through processing of rules, or that can pass through membranes based on certain conditions. Membranes can be dissolved or eliminated based on certain contexts. In a P System, computation progresses until the system halts, that is no further evolution of the objects is possible. At that time, the computation is complete.

Sets of objects, known as "multisets", are placed inside the regions depicted by the membranes. The objects are represented as symbols over an alphabet. Sets of rules in each region allow the objects to evolve to produce new objects from the existing ones. Objects can move from one region to another across a membrane if a rule exists to allow that. The ABC Machine is a derivative of a Membrane Computer, with string processing to more easily enable cognitive computing applications. The description follows in the sections below.

1.3 Sample Algorithms

Before defining the ABC Machine architecture, several algorithms and algorithm classes are discussed for illustration and analysis purposes. These examples serve to illustrate processing concepts with the ABC Machine, and to define the instruction set and operating system support needed to accomplish them. The examples were chosen that best fit the kinds of computing deemed to be the strength of the ABC Machine such as string manipulation, sorting, pattern matching, production systems and algorithms related to "cognitive computing".

<u>Binary String Matching</u>

Consider a linear binary string of 0s and 1s such as 000111001100. This can be thought of as a representation for a one dimensional signal or pattern. Let us look at both non-redundant pattern matching and redundant pattern recognition. The latter is useful when the signal is in the presence of noise. To match a specific pattern exactly, let us consider a shorter string, say 101. One approach is to have regions that are nested like a tree, as follows in Figure 2 below.

Each region contains a strip and move operation. The rule is to remove the rightmost atomic element, discard it, and move it to the next lower region depending upon whether the stripped atom was a 1 or a 0. If a single copy of the original string enters the ABC Machine, it will eventually "arrive" at a region that identifies it. This is the terminal state and therefore the solution to the pattern matching problem. Now consider the previous problem when there is noise in the pattern. Suppose we are trying to distinguish between X=000111001100 and Y=111111001111. If the "signal" is noisy, we might be offered the pattern Z=101111001111 and then attempt to determine which pattern it best matches. The algorithm above will not work because it is based on exact matching. Instead we need a set of rules that is tolerant of fuzzy matching. This is a problem where the statistical aspects of an ABC Machine may be applied.

One approach is to use a similar approach to the one above, but where the outermost layer has a rule that does an arbitrary flip of a 0 to 1 or visa versa for each position. Then begin by entering 100 copies of the string Z to be recognized. Allow these copies to "roam" around in the outermost layer (this is effectively accomplished in different manners depending upon the physical implementation). Eventually, each copy may run into an arbitrary flipping action. An equal number of rules in the outermost layer is a move operation that pushes the string to the next layer below. From there and below, the algorithm is the same as the exact pattern matching. In the end, the region that gets the most (a final counting algorithm is needed somewhat like discussed above) represents the answer to the recognition problem.

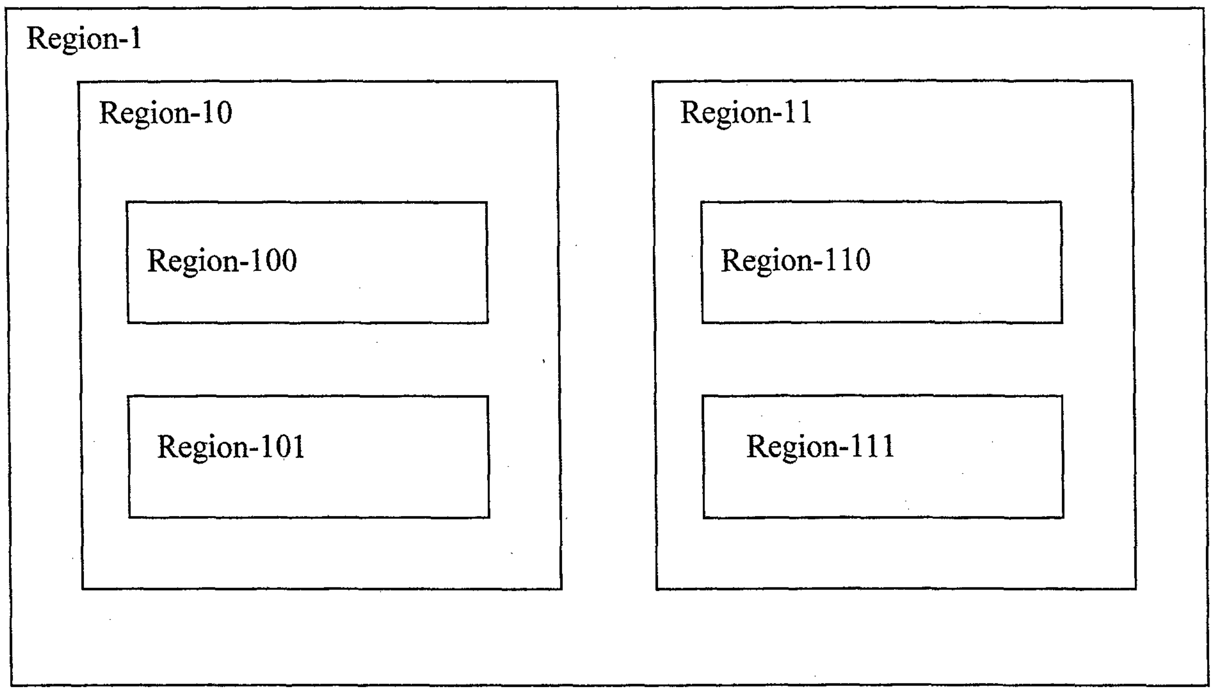


Figure 2. Nested Regions.

Sorting

There are many methods of sorting. One simple method that takes advantage of the statistical nature of the ABC machine is to begin with an initial state consisting of a large number of copies of each string to be sorted, say 1000. These copies are distributed randomly throughout a region of an ABC Machine. A large number of copies of the first compare rule above are also distributed throughout the ABC Machine in its initial state. However, the rule is modified so as to disallow any string from being operated on more than once. For example: s1, s2, t, u $\rightarrow$ ts1, us2 if $|s1| < |s2|$, else us1, ts2

This rule may not operate on any string beginning in t or u. Then apply a counting rule to the strings ts_i where s_i is the i-th string to be sorted. The highest count is the shortest or lowest number in the sorted string. This has the disadvantage that it requires arithmetic. A better approach would be to use nested regions and to move the shorter sequence to a lower level and have the ABC Machine nested n levels deep where there are n sequences to be sorted. The resulting final state is the strings. For sorting, we assume that comparison methods already exist (i.e. comparator instructions).

We examine two additional approaches to sorting and use them to illustrate important features of the ABC Machine, including the nature of convergence of statistical computing algorithms.

Sorting Method 1: Nested Regions Sort

The ABC machine/statistical dataflow process depends on multiple copies of data and on the moving of data based on operations on it.

Normally, parallel approaches to sorting are limited by the fact that an element in a list (or set) can only be in use at a single point in time. At most, then, when there are N elements to be sorted at most N/2 comparisons can be made simultaneously. In the ABC machine, X copies of each data element are made and similar elements in the list are propelled to the 'correct' placement 'in order' aggregately at a much faster rate.

The premise is this: all of the elements are contained in a single region and each element is duplicated X number of times to take advantage of (X • N)/2 simultaneous comparing operators. As two elements in the same region come in proximity they are compared. The smaller 'atom' is pushed into an interior region while the larger remains in the region. If the atoms are equal nothing happens. If there is no interior region one is created. Interior regions are singly nested. If an operation removes the last atom from a region, the local nested region's boundary is dissolved.

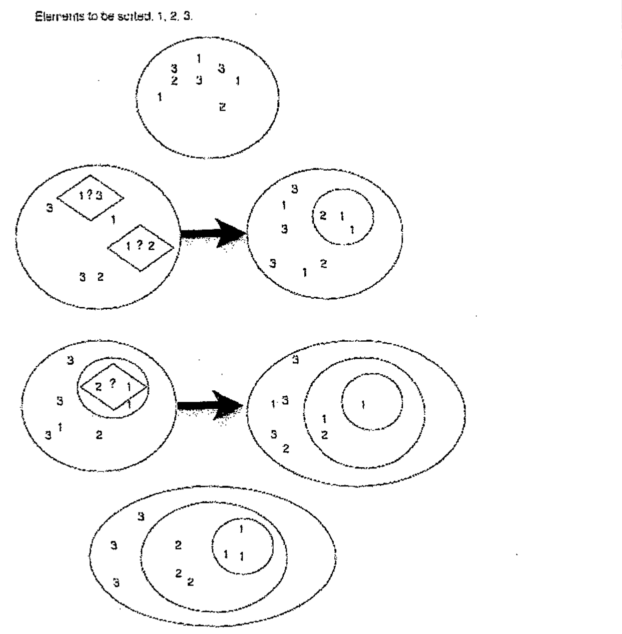


Figure 3. All concurrent operations not shown.

The advantage of the combination of numerous duplications and massive parallelism is that the 'correct' order of the elements is achieved very quickly and can take advantage of more simultaneous comparisons then there are elements in the set. The disadvantage is that resultant ordering only shows symbolic ordering. Most meaningful descriptions of the set are lost.

Sorting Method 2: "Fuzzy" Stochastic Cellular Progressive-State Method

Because of the massive concurrency of the state machine, 'fuzzy' sorting time can get better as N grows very large. This method maintains the integrity of a set of elements at the expense of the number of simultaneous comparisons. At most, only N/2 elements can be compared and re-ordered at any moment. However, the current state of the machine at any point in time is progressively better than the last. There is no guarantee that the list will ever reach 100% 'correctness', however, a close, but fuzzy solution is acceptable for the ABC Machine because of its objective to solve 'cognitive computing' problems.

Fuzzy sorting applies comparisons with contextual values and requires a more complicated string operations instructions set.

To sort, we need:

- The list of elements to be sorted
- At least N/2 'sorting units'

The Fuzzy Sort Method may be represented as:

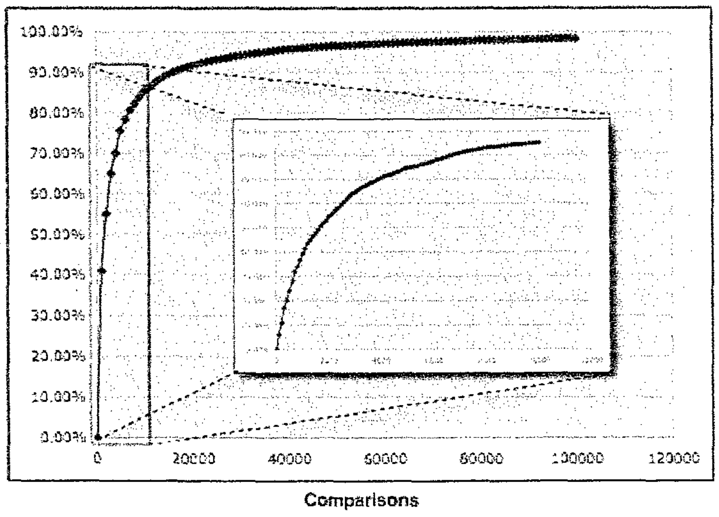
```
While list is still being sorted:
    1. Each sorting unit randomly attaches to two places on the list not already attached to a sorting unit.
    2. The elements are switched if they are in the wrong order.
```

Deterministic Hybrid 'Bubble' Method Addendum:

In the bubble method, a comparison unit consists of a 'head' and a 'tail'. N/2 comparison units are organized in a chain of length X. The head of the chain starts at the 'front' of the element list to be sorted. As each comparison unit engulfs an element of the list it compares the value at its head with its tail and switches the values if they are not in order. The comparison chain then proceeds to engulf another element making its way towards the end of the element list.

Each time the kth bubble comparison unit reaches the end of the list, the kth-1 element in the list is then in the correct position. The sorting pool consists of both random-attaching switching units and 'sorting chains.' The sorting chains from the deterministic method are not connected and only randomly find the 'front' of the list and make its way towards the 'end' of the list. The graphs in Figures 4 and 5 below consider 1,000 units and illustrate the 'convergence' of a statistical algorithm using an ABC Computing architecture. The benefit of the hybrid method is that elements can moves move quickly towards their correct position by being randomly compared (coarse ordering) and then nudged into their exact correct position by the sorting chain unit and guarantees a 'correct' order in much less than serial N^2 time.

Figure 4. Sort Comparisons.



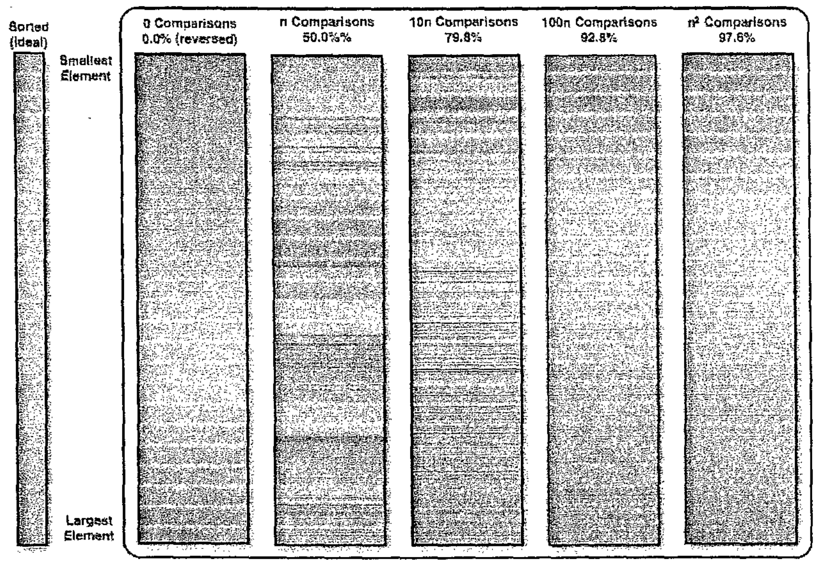
Sort 'correctness' is based on N-element lists with N different elements. The 'in-order' arrangement of the list has each index equal to its value. Percent correct is calculated as:

$$1 - \left[\frac{2 \bullet \sum_{k=0}^{n-1} x_k - k}{n^2} \right]$$

Comparisons	"Sorted"ness
n	~50%
10n	~78%
100n	~92%
1000n	~98%
10,000n	~99.8%

While not as efficient as a traditional sorting method, a large number of simultaneous comparisons could sort extremely large datasets 'very well' very fast. There is no copying of the element list required.

Figure 5. Graphical Sort vs. Number of Comparisons.

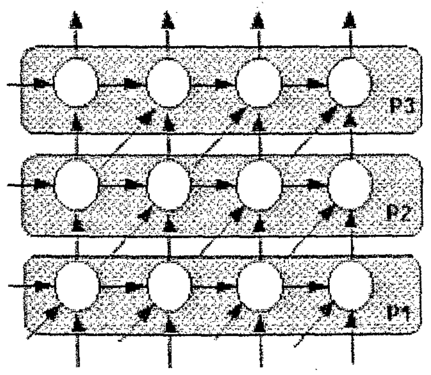


Graphical representation of 1,000 element list order. Green lines represent smaller values, red represent larger values.

Linear Equations

Can linear equations be solved using an ABC Machine? There have been numerous "dataflow" solutions to the solution of a system of linear equations in N unknowns in the field of signal processing as well as in the area of systolic arrays. Figure 6 below shows the structure of a linear equation systolic array. While these solutions are elegant in nature, they fundamentally differ form the ABC Machine architecture is two ways. First, the dataflow follows a highly organized path from one processing element to the next rather than a random path. Second, the processing elements require multiplication, addition and sometimes division. These arithmetic operations require an instruction set that goes beyond the scope of what is suitable in an ABC Machine which is primarily a string processing, redundant, statistical computing device. For these reasons, linear equations do not appear to be a good class of problems for an ABC Machine. However, linear equations are not generally at the core of cognitive computing tasks, and hence this is not a major setback.

Figure 6. Systolic Array.



1.4 Significance to Cognitive Computing and Benefits

DARPA defines cognitive computing as computing that can "reason, use represented knowledge, learn from experience, accumulate knowledge, explain itself, accept direction, be aware of its own behavior and capabilities as well as respond in a robust manner to surprises". Computer programs such as production systems, pattern recognition algorithms, and learning algorithms falls into this realm. The results of the Phase 1 study and the forgoing discussion showed that the ABC Machine as implemented on a HPC can perform computation in these areas of computing both faster and of higher quality than traditional von Neumann implementations of traditional AI programs. Some of the specific benefits of the ABC Machine architecture implemented on a HPC are itemized below:

- The ABC Machine can perform pattern recognition tasks well. It shows the promise of performing superior to traditional connectionist architectures for certain classes of problems that can take advantage of computational redundancy, more powerful instructions at each compute "node", and the computational power of a large scale high performance computer. We have shown the ABC architecture to be effective on dirty data, very large numbers of classes in classification problems, and it requires no training (i.e., it learns via adaptive unsupervised learning to an unknown number of classes).
- The ABC Machine is capable of performing symbolic computing tasks due to its string processing architecture. We have shown the ABC architecture to be effective on very deep search spaces and with heuristics for making "best first" decisions. The redundancy and parallelism enables such computing tasks to progress quickly. The redundancy enables symbolic computing tasks to proceed without the brittleness at the boundaries of traditional AI symbolic computing tasks.
- The highly parallel nature of the HPC implementation of the ABC architecture enables large scale cognitive computing tasks to be implemented, such as face recognition over a very large population, or such as automatic web service choreography – a complex planning task of a huge number of data items.
- Programming the ABC Machine is a simple as specifying regions, rule sets and atomic objects. The rest of the problem is a matter of parallel redundant processing. Hence, it does not suffer from software engineering complexities the way traditional computing does. In addition. machine learning tasks can be performed using the genetic programming approach to programming the ABC Machine.
- Overall, the ABC Machine implemented on a HPC provides an alternative for cognitive computing tasks that offers either substantial computing power, or better algorithm performance due to its statistical computing nature, or both. Problems that are difficult on conventional machines, such as pattern recognition, deep search space production systems, or complex system simulation, may be well suited to the ABC Machine.

We believe that the ABC Machine (or cluster of such modules) as described above and defined in more detail below provides an ideal computing architecture for cognitive computing tasks for the following reasons:

- Computing operations over tokens and strings of tokens is highly suited to symbolic computing, production systems and logic problems. In Phase 1, we demonstrated "best first" search using a combination of the A* algorithm and a shortest path algorithm implemented in an ABC Architecture on AFRL's high performance cluster. See below for details.

- A traditional neural network can be implemented inside an ABC Machine (or cluster of such modules) by defining a node (region or membrane) as a neural computing node, by using I/O rules to pass data to other nodes, and by using rules to do the combination of data (merge tokens into larger strings, for example). However, an ABC Machine is substantially more general (due to rule flexibility) than a neural net and so intuitively one could claim an ABC Machine can perform pattern recognition at least as good as a neural network (and probably better). In Phase 1, we demonstrated character recognition implemented in the ABC architecture on the AFRL high performance cluster.

- The statistical redundancy (many copies of tokens and rules) and random selection provides the potential to remove the brittleness of traditional AI by providing a voting or majority rule or multiple paths capability rather than a single failed rule leading to a dead end. The "best first" search algorithm continues to run when dead search paths are found because of this redundancy.

- It was modeled after biological processes and natural cognition is based on such distributed, statistically redundant paradigms, therefore the potential for new algorithms and more powerful reasoning exists with this architecture.

The field of Artificial Intelligence has emerged since the 1970s with various computing problem classes. Many of them follow the lines of symbolic computing that is largely based on logic, search and symbol manipulation.

Expert systems are programs that capture expertise and store them as facts and inference rules. To study the computational properties of expert systems, it is necessary to capture a description of them more formally. The approach usually taken is to describe an expert system more formally as a production system that is generated by context sensitive rules (Charniak, 1986). For example, the following abstract rules typify a portion of an expert system:

Rule 1: A → B
Rule 2: A and B → C
Rule 3: B and not A → D and E
Rule 4: C or F → H
Rule 5: D and (H or I) → J and K
Rule 6: H, J and K → Not B and Not D and Not E
Etc.

These rules are in the form of logical expression. Each context sensitive input (left hand side) implies an unambiguous conclusion (right hand side). A production system begins by a statement of logic or list of predicates such as: B and F, which can be described as state {B, F} which is the set of true predicates.. From here, the rules begin to process by forward chaining (Charniak, 1986) with steps such as follows:

Input: B and F, initial state is {B, F}
Rule 3 → D and E. New state is {B, D, E, F}
Rule 4 → H. New state is {B, D, E, F, H}
Rule 5 → J and K. New state is {B, D, E, F, H, J, K}
Rule 6 → Not B and Not D and Not E. New state is {F, H, J, K}

If this were the end of the search, the answer or result is the set of predicates {F, H, J, K}. The same result can be achieved through other search strategies such as backward chaining (Charniak, 1986).

For the purposes of an ABC Machine implementation, production systems can be thought of as agent based models (ABMs). A production system would potentially involve a large multiplicity of copies of the initial state in the form of string objects for each predicate. The agents, which are instructions would then execute in multiplicity.

There is no guarantee that an expert system consisting of ABC Machine instructions will converge, nor can a time estimate be made. The path through the rules leading to a conclusion may vary with the problem and the input data. This is the equivalent of a large scale agent based model (ABM) with no guarantee there is an attractor, limit cycle or limit point. It may converge or it may be chaotic. This is not different than an expert system implemented on a von Neumann machine, which are known to be brittle in performance. The improvement offered by ABC Machines is that results are fuzzy because of the large number of copies of each object. Some objects may have reached a conclusion while others may have taken a dead end path in the search space. Due to the presence of a large number of copies of each operand, if there is a path, some (and hopefully the largest concentration of) the objects will eventually reach that solution. The concentration of the solution objects will be less than 100% because of statistical processing, but in a good production system design, it will be the dominant answer for all possible search paths. This is a computability advantage for ABC Machines by introducing fuzziness or soft computing into the design.

For a system with N rules, M operands and K copies of each rule and L copies of each operand, the storage required is KNr + MLb where r is the size of a rule in atomic units and b is the size of an operand in atomic units on average. If the system is dominated by a large number of rules, the storage is proportional to N. If it has a small number of rules and is dominated by operand concentrations, then the space requirement is driven by the MLb term, which is independent of program size.

In our Phase 1 effort, we programmed an ABC architecture on the AFRL high performance computer and implemented two algorithms that fall in the "cognitive computing" realm. As mentioned previously, these were "best first" search and character recognition, both described later in this proposal. The success of these algorithms led to the selection of two Phase 2 applications (see Sections 2 and 3).

1.5 Detailed Computational Example #1 – "Best First"

Pathfinding or shortest path algorithms are computer algorithms that find a way to get from one place to another (called the start and the goal or end, respectively). Most such algorithms deal with graph representations of the problem space which store the cost of each node. These algorithms try to find a path along the graph with the lowest total cost, or distance. Another goal of all these algorithms is to find this path as quickly as possible while using as little memory as possible and still find the best path. There are many such algorithms. In cognitive computing or AI applications, depth-first search, breadth first search or best first search is often used. Our objective is to explore how to map a best-first search problem onto the ABC Machine architecture. But we shall first do this with a shortest path algorithm, as this is related best-first as described below and is a good algorithm to illustrate the ABC Machine features.

The simplest approaches are to go toward the goal until some sort of obstacle is reached then turn in another direction, and tracing around the edges of obstacles. This is an example of a "blind-search", where the algorithm does not rely on any information about the cost of the path to the goal in selecting the next node to expand. A list of other algorithms in this "blind-search" group are the breadth-first search, the bi-directional breadth-first search, Dijkstra's algorithm, depth-first search, iterative-deepening depth-first search. There are also some algorithms that plan the whole path before moving anywhere. Best-first algorithm expands nodes based on a heuristic estimate of the cost to the goal. Nodes, which are estimated to give the best cost, are expanded first. The most commonly used algorithm is called A* (pronounced A star), which is a combination of the Dijkstra algorithm and the best-first algorithm.

The different algorithms work in different ways. The breadth-first search begins at the start node, and then examines, or expands, all nodes one step away, then all nodes two steps away, then three steps, and so on, until the goal node is found. This algorithm is guaranteed to find a shortest path as long as all nodes have a uniform cost. The bi-directional bread-first search is where two breadth-first searches are started simultaneously, one at the start and one at the goal, and they keep searching until there is a node that both searches have examined. The final path found is then the combination of the path from the start to the intersection node, and the path from the goal to the intersection node. Dijkstra's algorithm looks at the unprocessed neighbors of the node closest to the start, and sets or updates their distances (in terms of cost, not number of nodes) from the start. The Dijkstra algorithm expands the node that is farthest from the start node, so it ends up "stumbling" into the goal node just like the breadth-first search; it is guaranteed to find the shortest path. The depth-first search extends nodes (it extends a node's descendants before its siblings) until it either reaches the goal or a certain cut-off point, it then goes onto the next possible path. The iterative-deepening depth-first search is like the depth first search, but the cut-off point starts at the straight line distance to the goal, and once all nodes up to that point have been expanded the cut-off point is incremented and the search is run again. The best-first search algorithm is a heuristic search algorithm, meaning that it can take into account knowledge about the map; it is similar to Dijkstra's algorithm, but it goes to the node closest to the goal, instead of the node farthest from the start.

The A* algorithm works much like the Dijkstra and best-first algorithms only it values nodes in a different way. Each node's value is the sum of the actual cost to that node from the start and the heuristic estimate of the remaining cost from the node to the goal. In this way it combines the tracking of previous length from Dijkstra's algorithm with the heuristic estimate of the remaining path from the best-first search. The A* algorithm is guaranteed to find the shortest path as long as the heuristic estimate is admissible (an admissible heuristic is one that never overestimates). If the heuristic is inadmissible then the A* algorithms won't find the shortest path (or a path at all), but it will find a path faster and using less memory. While the heuristic must never overestimate, the closer it is to being correct the more efficient the A* algorithm will be; in fact, the Djikstra search is an A* search, where the heuristic is always 0. This algorithm also makes the most efficient use of the heuristic function, meaning that no other algorithm using the same heuristic will expand fewer nodes and find an optimal path, not counting tie-breaking among nodes of equal cost. One of the problems of the A* algorithm, as well as many other pathfinding algorithms, is that they take up a large amount of memory by storing all previous nodes or all previously take paths.

Shortest Path Algorithm and Best First Search

We shall first explore specifically the shortest path algorithm as it is structurally similar to the rest as discussed above. ABC machines could be useful in the resolution of shortest path. The algorithm we evaluated in Phase 1 works more like sending a bunch of "rats" into a maze and finding which rats utilize the shortest distance to get to any particular node (blind search with multiple copies taking advantage of the ABC redundancy). The components and the algorithm are described below.

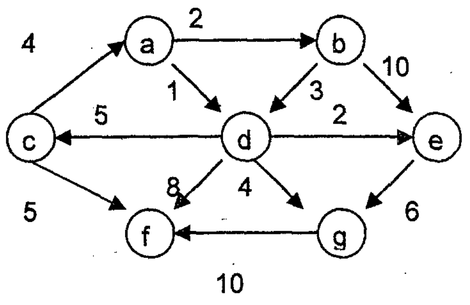
The algorithm for shortest path in the ABC Machine implementation has tow process components. The first is a "Path Discovery Cycle" that contains the "rats" that crawl through the graph and find paths to its various nodes. (The "rats" are represented by strings). The second is the "Path Comparison Cycle", that compares reported paths to various nodes in the graph and consistently maintains record of which are the shortest for any given node. Figure 7 below illustrates the algorithm.

Vertices

The graph would be represented by its vertices. Weight is recorded by repetition of the symbol corresponding to the destination. The first character always refers to the node while the latter ones show what paths are available from the given node.

adb^2
bd^3e^{10}
ca^4f^5
$dg^4c^5f^5e^2$
eg^6
gf
f

(note: super script represents repetition)



Graph Borrowed From Data Structures and Algorithm Analysis in Java, Author Allen Weiss, 1999, 0-201-35754-2, page 304

Figure 7. Shortest Path Graph.

In this representation of shortest path, **Ta** Represents our exploration "rat" with starting node "a". The purpose of the symbol **T** is to prevent the system from confusing the exploration "Rat" with a vertex. There will be N copies of **Ta** introduced into the system as well as N copies of the vertex mappings. This ensures the proper redundancy and resources for the system to evaluate the shortest path. Now, we explore these two cycles in more detail.

Path Discovery Cycle

As the "rat" (who is one of many) floats around in our region filled with vertices it will eventually find a vertex that has an initial character identical to that of the "Rats" ending character. Initially, with the vertex mapping provided in the illustration above, the first match to be made will be with a "rat" **Ta** and copy of vertex $\mathbf{adb^2}$.

1. We break first the vertex apart into **a d bb** where "a" is a node which goes to **d** with a cost of 1 and **b** with a cost of 2. *(This part could optionally be omitted if we enter in all vertices individually instead of in groups…the purpose of the groupings here are to encourage exploration of the optimal choice first as can be seen below in step b).*

   a. Next we repel any path containing destinations that are already in the string…(on this first comparison there will not be any for this graph)

   b. Second, we then compare which path is shorter **d** or **bb**. We conclude that **d** is the shortest and append that path onto our "rat's" string.

   The conclusion of this process results in our exploration "rat" holding a path of **Tad**

2. We then **copy** this discovered path to the path comparison cycle for further evaluation with the results reported from other "rats". The original copy is left in the discovery cycle to seek other matches and repeat the above process.

3. To help ensure that the optimum choice is taken at each step we can reassemble the remaining paths from that node for future iterations of step 1. In this example there was only one other path (b^2) for the initial comparison, so we simply append an "a" in front of it and leave $\mathbf{ab^2}$ to be explored by another exploration "rat".

We then continue this cycle with our "rat" of path of Ta**d** as it encounters vertex $\mathbf{d}g^4c^5f^5e^2$ (or any other string starting with symbol **"d"**).

Path Comparison Cycle

The next part of the processing is the path comparison cycle. Here we take the two stored paths:
Tad – From example above $\mathbf{Tab^2d^3}$ - from another concurrently explored path

The process proceeds as follows:

1. Look for paths with same ending character and compare.

2. Path with shortest distance (I.E. fewer characters) survives.
   (note: remember that the superscript represents repetitions of a character)

   Tad has fewer characters than **Tabbddd** therefore **Tad** gets the vote over $\mathbf{Tab^2d^3}$ and $\mathbf{Tab^2d^3}$ gets destroyed and its resources are released back into the path discovery cycle.

The cycle will continue until the process is halted.

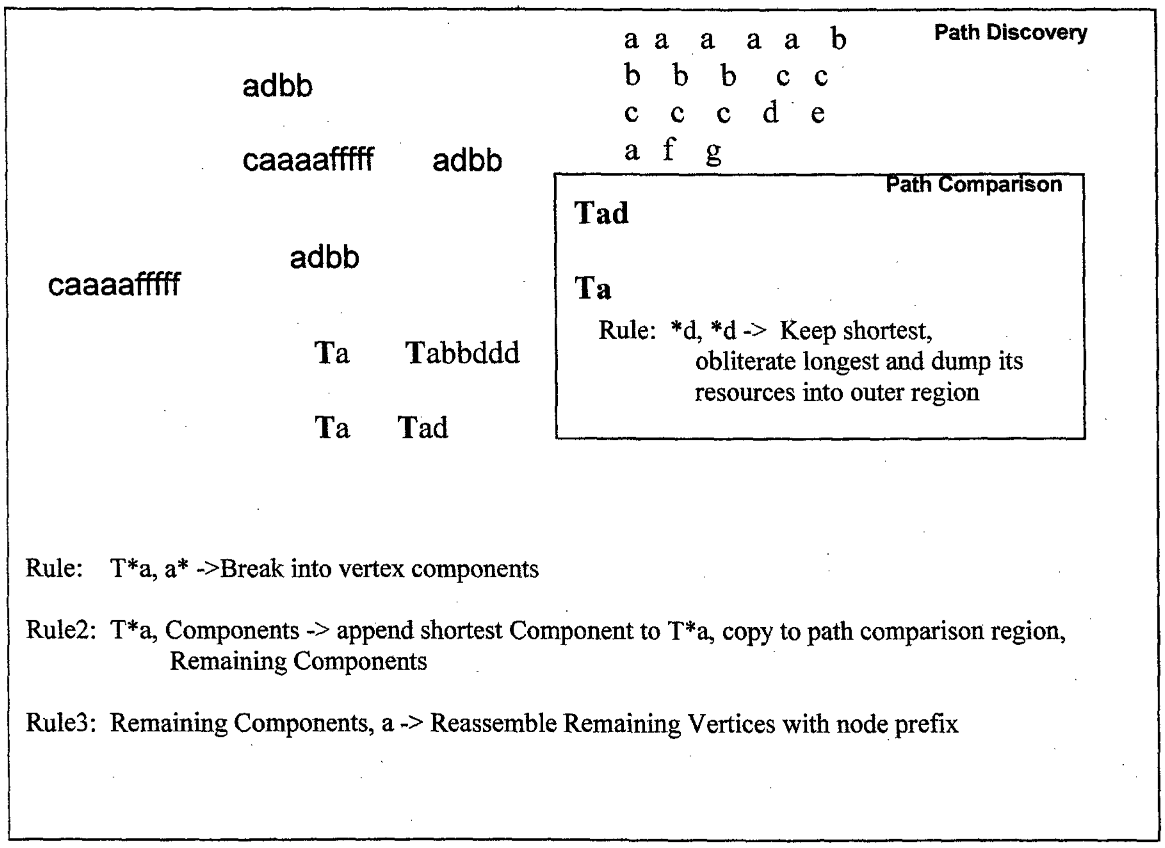
Our result may be obtained by looking at the only remaining values inside the path comparison cycle with the desired node on the end.

*For example: our result for shortest path for node **d** is **Tad***

Mapping of Shortest Path Algorithm to ABC architecture

In this architecture, the described "Cycles" would actually be mapped to regions of the system. The outer region would contain our path discovery cycle and would be filled with N copies of all the vertices as well as N copies of our "rats". Also in this region would be a variety of miscellaneous symbols required for generating copies send to the inner region containing our path comparison cycle.

Figure 8. Illustration: Example of ABC regions and contents



Rule: *d, *d -> Keep shortest, obliterate longest and dump its resources into outer region

Rule: T*a, a* ->Break into vertex components

Rule2: T*a, Components -> append shortest Component to T*a, copy to path comparison region, Remaining Components

Rule3: Remaining Components, a -> Reassemble Remaining Vertices with node prefix

Applications of the Shortest Path Algorithm Methodology

The procedure below shows how to apply the shortest path method. Utilizing the shortest path algorithm method:

1. Must generate the vertices of the graph
   a. This can be done ahead of time (graphically specifying all possible states that can occur in a graph).

b. This can also be done by generating new vertices "on the fly" dynamically producing the graph as we move along. This would mean our graph is in a constant state of development, and is not the same from one nanosecond to the next as it is added upon. This would require a change in the defined procedure for step 1 of the cycle be altered so that instead of breaking the given string down into its potential vertices we produce the new vertices based on a heuristic.

2. We must also come up with a means of determining which path is the optimal choice at each given state. *Recall the shortest path example where we always attempted to follow the vertex with the smallest weight.*

3. Although we are applying concepts from the shortest path algorithm listed above, this does not necessarily mean we have to look for the *shortest path*. Alternatively we could introduce other criteria for which to compare and eliminate other than that of path length.

4. Although the shortest path example used symbols for nodes and their connecting paths, we can replace these with strings enclosed with terminal characters if necessary to help in a variety of other applications where a single symbol wouldn't be appropriate.

This example uses a method of generating vertices dynamically as it encounters a vertex pointing to itself. Our goal is to find a path that will get us to our goal (or to the heuristically closest solution we can find).

Vertices	Initial State
X represents blank slot and U is a separator to discern state representations. 283164X75**U**2831647X5	Goal State

2	8	3
1	6	4
7	X	5

1	2	3
4	5	6
7	8	X

Figure 9. Exploration path T2831647X5

Path Discovery Cycle

The next phase is the path discovery cycle, explained by the procedure below.

1. Trailing string of exploration path is matched with leading string in vertices.
   a. If **encounters a node with a vector to itself**, generate potential vertices from this node and corresponding self pointing vertices for further subsequent generation of graph.

   b. Otherwise jump to step C (IE encountered a vertex not pointing to itself, hence do not generate new vertices)

Initial Vertices	Resulting Vertices
2831647X5**U**2831647X5	2831647X5**U**283164X75**U**28316475X **U**2831X4765 283164X75**U**283164X75 – for further graph generation 28316475X**U**28316475X 2831X4765**U**2831X4765

c. Pursue which path is determined favorable if any (best first optimization). For simplicity of our illustration we will assume that the first path from the node (marked in red) is the heuristically preferred path.

Vertex (before): 2831647X5**U**283164X75**U**28316475X **U**2831X4765

Update Exploration Path: **T2831647X5U**283164X75

Vertex (after): 2831647X5**U**28316475X **U**2831X4765

d. Vector is left for other exploration paths to discover in parallel
e. Cycle is repeated continuously as the "rats" wander through our graph

2. Copy Exploration Path to **Path Comparison Cycle**
   a. Compare strings on end of paths (component following last **U**). Keep path with fewer discrepancies in the ending string based on our goal, destroy the other. Resulting condition will be finding a solution closest to our goal until we actually find it.

Summary --Best-First or Breadth-First?

The approach of shortest path discovery by the ABC machine is to define the connectness of the graph and disclose the starting points of the desired path and let the machine go. The more starting points you allow the machine to begin with, the more diverse short-trip discoveries can be made and added to the collective.

With Respect to graph traversal, the ABC machine exhibit swarm like behavior with a very large number of agents independently working unique solutions with the added benefit of the collective's past successes. Unlike swarms, however, the ability of each individual agent can be and is much more complex and flexible. Additionally, the ABC has the ability to let each agent contribute to a global memory architecture that maintains shortest paths found by the collective.

Future implementations could subdivide the problem based on a set of rules and then investigate the sub-problems in Multi-swarm fashion the difference being that each split could approach the problem from different ends/starting points yet still contribute to a central memory.

This type of behavior seems ideal when exploring large search spaces. Some particles will explore far beyond the current minimum, while the population still remembers the global best. This seems to solve one of the problems of gradient-based algorithms.

The above algorithm would by its nature be best classified under a parallelized breadth first approach. With the optimization, it could begin to take on the nature of a best first search, as it would be adding search optimization unto a breadth first system. Therefore rules could be introduced such that our "rat" would decide which direction to take based on things such as hamming distance or any other criteria that would be acceptable to a particular problem. The result could be a search that behaves more as a best first search where it would look for the best choice toward the goal at each comparison.

The shortest path and its potential breadth first search implementation were presented in sufficient detail to establish the case for the applicability of symbolic computing paradigms to the ABC Machine. This section will be referenced as the system design evolves.

1.6 Detailed Computational Example #2 – "Pattern Recognition"

Pattern recognition is a class of problems that traditional von Neumann architectures do poorly and neural network architectures do reasonably well. It can be argued that ABC Machines should be able to do pattern recognition very well, and at least as well as neural networks. The reason for this is that neural networks can easily be implemented in an ABC Machine, but the architecture of an ABC Machine is a much more powerful generalization of neural networks. Why is this the case? Each node in a neural network could be implemented in a ABC Machine region with a more general instruction set than the simple additive mechanism of neural nets. The network could be "programmed" through suitable pathways to implement a neural network node integration. The network connections could be formed using directed I/O instructions.

For example, a neural network node received inputs from multiple other nodes and integrates the results into a single value that represents a signal level. This can be tracked by appending a token to a string corresponding to the level of stimulation, and then systematically digesting the tokens off the string as a function of time to diminish the signal. For example, consider the token C (for carrier) and the token S (for signal). An integrating instruction can be represented as:

$$(S_1S_2..S_NC,\ S_1S_2..S_MC) \rightarrow (S_1S_2..S_{N+M}C,\ C)$$

which yields a larger signal string $S_1S_2..S_{N+M}C$ that might possibly be exported to the next node. At the same time, a digestion instruction insures that these strings do not last long and exists in concentration as:

$$(S_1S_2..S_NC) \rightarrow (S_1S_2..S_{N-1}C,\ S)$$

The export of strings to the next node takes place using an I/O instruction of the form:

$$(S_1S_2..S_NC\ ||\ ) \rightarrow (\ ||\ S_1S_2..S_NC)$$

This moves strings, short and long, to the next node which can then measure signaling strength by the value of $< S_1S_2..S_NC >$ for different N.

Another mechanism for pattern recognition with the ABC Machine can be illustrated by considering the problem of the recognition of characters (A, a, B, b, ...0,1,2,...). It is motivated by the binary image processing techniques presented by Horn (1986) which aggregate dark spots and count features of the resulting shape. Figure 10 below shows a simplified version of such an image.

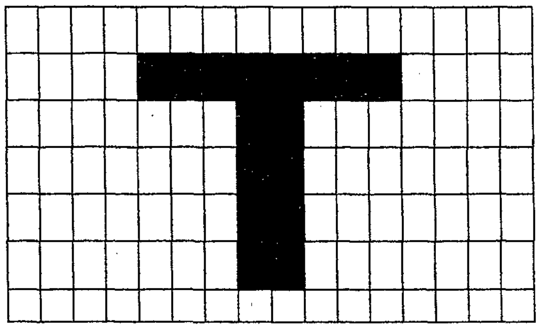
A tree can be used to represent the image as follows:
{ (0 0 0 0 0 0 0), (0 0 0 0 0 0 0), (0 0 0 0 0 0 0), (0 0 0 0 0 0 0), (0 1 0 0 0 0 0), (0 1 0 0 0 0 0), (0 1 0 0 0 0 0), (0 1 0 0 0 0 0), (0 1 1 1 1 1 0), ...etc...}

Many copies of this tree may be passed to an ABC Module for evaluation. This is due to the statistical processing requirement. The way it works is to aggregate regions into three layers or processing regions.

The first layer is the "reader layer" which is fed a binary string representing black and white for 0 and 1, each of which is an image pixel making up a part of the character. The second layer is a "processor layer" that counts the instances of neighboring ABC Machine or region that match the string values contained for each ABC Machine region by keeping an expression level of strings of various lengths. It can "ligate" strings of various lengths in a given neighboring direction. The third layer is a "processor layer" which operates on ligated strings and counts directional changes by expression levels. For example, the A and B characters will result in dramatically different expression levels (an "expression level" is the vector describing the concentration of a list of operands in a region) and hence that serves to identify the character. Each character to be recognized is represented by a different processing region.

The different approaches show that many approaches can be evaluated for the best fit for a pattern recognition problem due to the high flexibility of the ABC Machine architectural concept.

Figure 10. Binary Image of a T.



The pathways (instruction sequences) that are used to process each character implement standard binary image processing algorithms (Horn, 1986) such as run length encoding (which calculates the length of a binary run of 1s), pixel counting (i.e., how may 1's are there), location of the centroid (as determined by thinning, resulting in a remaining remnant), etc. Horn (1986) provides many such algorithms and only experimentation (or genetic programming) can determine the best fit. The resulting "features" are ligated into a string whose concentration is evaluated by a sorting algorithm which could be implemented with digestion and ligation rules.

In fact, an ABC machine could be thought of as a generalization of a neural network. This is because a neural network can be implemented in an ABC machine architecture by allowing every node to be a membrane or region containing the neural integration function. Every connection in the neural network can be implemented through region I/O instructions sending results to other regions. While a neural net can therefore be implemented in an ABC machine, an ABC machine is substantially more general (and hence potentially more powerful) than a neural net.

Now, for some specific details on how to implement a character recognition solution in the ABC Machine using several methods will be presented. Traditional ANN classification systems work by training it: presenting a set of prototypical data to the network and letting the network adjust itself incrementally arriving at an optimal organization of weight-connected nodes that accumulate a vector. After the network has been trained, unclassified data is presented to the network producing a vector that then must be interpreted by the user.

In a large distributed statistical processing architecture the process is different. Training the network involves populating Identity Regions with large variations of the matchable patterns with each region 'knowing' about a particular category. In character recognition, training would involve describing the letter 'A', for example, in a variety of different typestyles using different orientations and scripts. All of these variations would be placed in a single Identity Region.

The Identity Region might also maintain 'Stream Metrics' that define the regional category strings in abstract ways like front/back weight, balance, and symmetry. Another Identity Region would be fed a variety of definitions of 'B's. A 'defined string' is described in terms of the a single character string built by sectioning an NxN matrix that circumscribes the character and evaluating each element at i,j as 'on' or 'off' (1 or 0) or by evaluating each section for graduations of 'on' (1,a,b,c,d,e,f,0, perhaps). There is no requirement that the Identity cell describe visible objects. Training can be done by traditional 'supervised training' methods but Epoch and back-propagation are unnecessary because the system already knows how each possible character is defined and does not depend of network weight corrections. The algorithm is outlined below.

Categorization Method

```
    while there are comparison units in the most nested region
        1. Each comparison unit has a reference to the subject to be classified.
        2. For each comparison unit: (implying simultaneity, NOT a loop)
        3. Two random elements from any two (including the same) Identity Regions are chosen.
        Selection can be done by proximity as the comparison units chaotically move around the
        region.
        4. The element 'more like*' the subject plus the comparison unit are promoted 'out' one level
        to an encompassing outside region. If there is not yet an outside region, one is created.
        end while;
    remove reference to nested Identity Regions
while still refining the categorization
    while there are comparison units in the most nested region
        1. Each comparison unit has a reference to the subject to be classified.
        2. For each comparison unit: (implying simultaneity, NOT a loop)
        3. Two random elements are chosen. Selection can be done by proximity as the comparison
        units chaotically move around the region.
        4. The element 'more like*' the subject plus the comparison unit are promoted 'out' one level
        to an encompassing outside region. If there is not yet an outside region, one is created.
        end while;
    destroy most nested region
end
```

The first 'pass' of comparisons are made from Identity Regions while subsequent passes are only made from promoted Identifiers. As time progresses, only the Identifiers that are more like the previous classifying subjects are left. It should be noted that comparison time is not necessarily linear and depends on the number of simultaneous comparisons that are able to be made

*For the purposes of this experiment, in all comparisons 'most like' will be determined by *Hamming Distance*.

The Hamming distance between two character strings is the number of positions in which the characters of the two strings are different.

As we get further into experimentation other possible string matching techniques like Edit Distance or Dice's Coefficient matching or for more biologically inspired techniques like Smith-Waterman-Gotoh distance or BlastP similarity tests might be considered. A cursory examination of a variety of string matching techniques can be found at http://www.dcs.shef.ac.uk/~sam/stringmetrics.html#hamming .

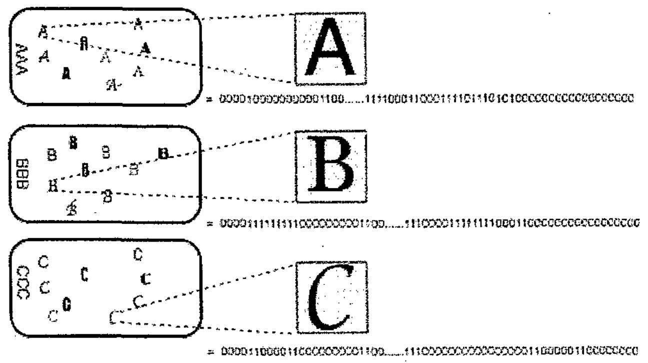


Figure 11. Shows concept of Identity Regions and string representation of characters.

We have previously discussed implementing pattern recognition on the ABC machine by encoding the data into strings of N characters in a methodical way. Using our example of encoding letters for character recognition, for example, we could use a 5x5 grid overlay on a recognized character and represent that scan with a string of 1's and 0's representing 'on' and 'off' bits when the majority of a grid point was occupied by the stroke of the character being classified.

Traditional character pattern recognition involves similar representations with each sector of the grid representing an input into the Neural Net. Each of those are then connected to n hidden-layer nodes and each of those are connected to an output with the total number of outputs equaling each possible classification. After feedback training, with a quality representation of a set of inputs and outputs, the network can classify even noisy and faulty input with a significant degree of success.

In the ABC machine implementation, however, no training is needed. The training data can instead be represented in the network as a set of stored 'memory' that is accessed and used for comparison with the subject being classified. Because there is no training involved, additional profiling data can be added to or taken away from regional 'memory' dynamically to aid in classification *while the machine is running.* Further, the restricting of inputs is lifted and additional classification criteria can be used to improved the accuracy of a solution.

For example, while comparisons to the bit string representation mentioned above are being used for classification, other evaluations of the string can augment the quality of selection. Just like humans use characteristic clusters to aid us in recognizing objects, the ABC machine can make more accurate comparisons by analyzing the data stream for "Metrics": groups of bits, pairings, string weight or density, color, runs of white or dark space, and perhaps even edge detections or the effect of specific mutations or operations on the bit stream and their result. Even more exotic matching metrics could be discovered by building and evolving the Metrics rule sets genetically or by incorporating probability or other statistics.

Simple character recognition using simple Hamming distance has been shown to be successfully even with noisy data even in examples when a human could not make a valid classification. In the future, the classification rule set will be expanded to facial recognition by including photographic metrics including color, contrast, heat points, and edges and for even human-observed characteristics like young, old, white, black, Asian, Hispanic, etc, plus potentially age-related metrics and automated age-progression technology (Ricanek, Patterson, Midori). Names don't matter because they only are used to define a string pattern that represents that characteristic: the ABC only uses that named stream to evaluate some measure similarity of it to the subject being classified.

During Phase 1, we successfully implemented a simple character recognition in the ABC Machine architecture with good results. With 100 initial comparisons (to build the first tier of 'good' guesses) the simple Hamming distance and nothing else gets it right even with noisy input. We used the characters A, B,C,D,E,F,1,2,3,4,5 & 6. The entire process ran in 0.54 seconds.

Basically, there has to be enough tier one comparisons to make the probability of getting a 'correct' guess very good. Even with only 10 tier one comparisons it got it right 8 times out of 10. The two times it didn't guess 'E' it guessed '6' which is not bad considering if it never saw an 'E'.
Also, we have realized that the Hamming distance only has to be calculated once for a given class, so that will save a lot of time when more complicated.

In summary, an overview of classification method in the prototype of the ABC Machine is outlined below:

- Classification objects are preprocessed to be represented in a stream of characters or atomic objects.
    - For example, to recognize a letter for character recognition a *nXm* grid is superimposed on top of the letter and grid squares that are predominately occupied by the stroke of the letter are represented as a '1' in a stream of '1's and '0's that is *n•m* bits long corresponding to the *n* rows and *m* columns in the grid.
- A bitstream that represents each desired class is inserted in to a matching region.

- For future classification enhancement it is possible to include separate regions for sub-classes ie: you could have a serif region and a san-serif region to improve making parameter distinctions that you already know about before classification. Another example is that each region could be a character trait of a human face: male, female, black, white, Hispanic, Asian, old, young, etc.
- Encoded in the stream or included with the object is the class identifier.

- Through a time limit or through a fix number of trials, the subject to be matched is compared with two randomly selected class bitstreams.
  - It is not necessary for them to be unique and in fact variety in the class descriptions could greatly improve the quality of class matching.
  - The stream of the randomly selected pair that is most like the subject being classified is moved into an upper 'Staging' Tier.
- After the time limit, the process is repeated on the upper staging tier except that instead of selecting comparison members from a class they are selected from the staging tier.
- The process can repeat until there are no more upper tier members left or until a fixed amount of time.
  - If comparisons have reduced your classifiers to a single bitstream then it is the most likely candidate for a match.
  - With a time-limited approach, it is possible to get a description of the class through the use of a frequency table of classified objects. Recall that the solution presented in the ABC machine is the state of the machine in time. If you evaluate the frequency of a class of streams in a staging tier then the profile of the streams is the expression of the likelihood of a future match. For instance, with a facial recognition example, if after 10 seconds you have made 100, 000 comparisons and have received 22% 'White' classification stream matches while receiving < 5% other class matches you can predict that the final recognized match will be white.
  - The chaotic nature of the ABC machine helps to generalize solutions because it finds matches without bias. The primary limitation on class matching is set size: you have to work to ensure that you have a high probability of grabbing enough classes to confirm a match hence the distribution of the problem over a large number of machines. However, even with the limitation you can likely still build a profile of the class of the subject

Some things to consider regarding our results for this exercise are:

- The ABC machine implementation needs no training. With a Neural Network, time must be taken to acquire quality matches and to adjust the network for a set of weights for future classification. Also, if there is another class of data to be added to the network in the future, the entire process of training has to be redone. The ABC machine has no such limitation: to add classes you just add them. It could even be done while the program is running!
- Neural Network classification is simple! There is no reason to limit matching classifications to simple stream matches (ie: Hamming distance, String matches, run-length encodings, stream metrics etc). Because the classifications are done by rule matching, there is no limit to the type of evaluations performed between types of classes. In the ABC machine, it is perfectly valid to determine class similarity by string matching in one case but by performing some mutation of the stream before comparing to another!
- By building frequency profiles you can get the same 'fuzzy' classification inherent in ANNs. Often noisy or 'outlier' inputs won't match to exact outputs but the network can present output in terms of percentages. Frequency tables provide the exact same benefit to noisy and/or completely alien input.

1.7 ABC Machine Formal Definition

An ABC Machine is a specialized form of an Agent Based Model (ABM). An ABM is a model consisting of multiple "agents", which are entities or processors that act on their own behalf with a set of rules. For example, the human immune system response is an ABM that is also an ABC Machine. Each cell type can be thought of as an agent or ABC Machine membrane. Each has their own rules of operation. Agent based models (ABM) may be ABC Machines, but in general are not. Not all ABMs have rules that follow a membrane computing paradigm.

A traditional von Neumann computer is not an ABC Machine. It is a control flow architecture and does not obey the principles of stochastic dataflow computing. An ABC Machine can be simulated on a traditional von Neumann machine to the extent that random numbers can be generated to represent the behavior of a large number of objects. Such a simulation would require a very large memory to account for all of the ABC Machine regions, objects, states and rules.

The instructions in an ABC machine are simple string operations. However, it was shown above that a combination of such instructions in a region can compute logic gates (or other logical computation). Hence, a region (which is a large set of instructions and operands) can act as an "agent". In the theory of agent based models, an agent is capable of making decisions and responding to context. The ABC machine then is capable of implementing agent based models and hence the name ABC machine.

The architecture of the ABC Machine is defined by enumerating its **computational features**, **baseline instruction set, memory characteristics**, and **operating system features**. This section enumerates these items.

Definition of ABC Machine.

An ABC Machine is a "P System" (a membrane computer as defined by (Calude and Paun, 2001) that also obeys the following additional restrictions. Rules represent a Context Sensitive Grammar (CSG) plus rules for adding and dissolving membranes. Certain definitions of terms are defined within the list and brief explanations are given. The list is as follows:

1) **String Objects**. Objects are string objects and the rules operate over string objects possibly changing their composition by combining parts of objects together or by splicing (i.e. digesting) them. Programs are stored as one or more strings of embedded operands and programs execute by arrival of activating or regulating operands. Programs do not execute entirely, but only those instructions that are instructed to become active do so.

2) **Conservation of Matter**. Matter is conserved. That is, the atomic elements of the string objects remain in existence unless they are removed by an I/O operation. Similarly, no new atomic objects may be created unless it arrives through an I/O operation or the digestion of a strong object.

3) **Enzymes.** Rules can contain enzymes. An enzyme is an object that is not modified by a rule but must be present for that rule to execute and may affect the energy needs of the rule execution (see next item).

4) **Conservation of Energy**. Rules may require energy tokens (represented as a specifically designated string token). An energy token is a token that is consumed or reduced in state when the rule executes. For example, 'appp' is an energy string token and it takes the form 'app' when the energy object p is consumed. The token app may be converted back to 'appp' by the addition of an atomic object p with an appropriate rule. Energy may not be created or destroyed in an ABC Machine, but must either be present in the form of adequate tokens, made available by combining tokens, or obtained through an I/O operation. Note that classical P Systems ignore this biologically relevant constraint.

5) **Statistical Processing**. Rule processing is statistical. They contain a very large number of copies of each operand and a rule, in general, operates many times on many copies of its operands and produced many copies of its outcome. Program execution is massively parallel, distributed, multi-threaded and otherwise contains no central processing unit.

6) **I/O Operations**. It must have I/O operations across all membranes.

7) **Context Dependency**. Rules (also called instructions) must be context dependent. The rule must be applicable to a membrane where it executes (not all rules apply to all membranes) and all operands, energy tokens and enzymes required by the rule must be proximal or randomly selected from those present. Rules may have decision making logic that respond to the environment (concentration of operands – the state vector), and bias or "mood" as established by its history of exection (example shown within the proposal).

8) **Dynamic Membranes**. Membranes may be created or destroyed by rules. Membranes are destroyed by dumping their contents into the next level membrane above it. Membranes may be created by forming a new membrane around a proximal subset within a membrane. For example, a rule to destroy a membrane may occur within that membrane per the following example: X, E, appp → X, E, app, Xmembrane. Here, Xmembrane means remove the membrane. It can also be done at the level of the next higher membrane: Membrane1, E, appp → X-Membrane1, E, app. It may cease to function by its own decision if it becomes defective.

9) **Programming.** The node may be 'programmed' by manually fixing the instruction set and operands or through genetic programming. It contains an operating system that provides the mechanism for copying instructions and operands from programs, delivering them to the proper location, and providing the power for their execution upon the arrival of their operands.

10) **Statistical Dataflow Computing.** Rules and operands are assumed to be in "motion" within their regions or membranes. There may exist multiple copies of rules and/or operands, perhaps a very large number. The "motion" need not be physical motion in a given physical realization of the ABC Machine, but it must represent a random shuffling of the rules and operands. When the operands for a given rule come into "contact" with that rule, the rule may fire. The term "statistical" implies many copies and the computing results are the aggregate concentration of resulting tokens. The term "dataflow" means that rules fire when operands "arrive".

Baseline Instruction Set of the ABC Machine.

Common to representative algorithms in the ABC Machine is a baseline set of rules or instructions. In general, rules or instructions may be any Context Sensitive Grammar (CSG) plus rules to add or dissolve membranes. The instructions that follow are those we have chosen for the baseline design of our ABC Machine implementation for this project. These common instructions are enumerated here, used later to illustrate computing algorithms.

MOVE. Communication is one of the most basic and important operations in an ABC Machine. The most basic communication rule is a "move" operator. An example of a move operator is the following:

R_k: E_k, A → E_k, -A

What this says is that Rule k executes when Enzyme k and the string A are present. The result of Rule k is that the Enzyme k is left unaltered and the string A is moved out of the region to the next higher region (Region 1) of the ABC Machine. If the rule has said +A rather than –A, then the movement is to the next lower region (Region 3), or the nearest such region if there is more than one lower region.

So if this rule were present in Region 2 above and it executed, the result would be that Region 2 would be left with only two A strings and a B string and Region 1 would gain an A string. This is a simple move operator and it could be thought of as just one instruction or rule among a large set that is doing computation aimed at solving a particular problem.

COPY. Sometimes it is necessary to produce a copy of a string. A rule to do this might look like the following:

E, ab $\rightarrow$ E, ab, ab

The problem with this rule is that it seems to produce strings out of nowhere. But the ABC Machine has both energy and mass conservation properties by definition. Hence to make a copy of a string requires that there are enough atomic objects available to make the new string and there is enough energy to execute the operation. For example, a more accurate statement of the rule for an ABC Machine might be:

E, a, b, ab $\rightarrow$ E, ab, ab

These notions of conservation of matter and energy come from biology. A large molecule cannot be synthesized by a cell unless smaller molecules that are its constituents (nutrients or amino acids) are available. Nether can it be done if adequate energy (i.e., ATP) is available. In our case, the energy can be tracked by requiring the presence of E in the rules above, even though it does not contribute to the production of strings.

LIGATION. Ligation is a basic string operation. It merges two strings. For example the rule:

abc, def $\rightarrow$ abcdef

is a ligation rule that takes two short strings abc and def and merges them into one longer string abcdef. The rule is very string specific and not algebraically general. It was discussed above. A more sophisticated version of ligation might be included to "insert" a character string in the middle of a string.

DIGESTION. Also discussed above was the digestion instruction. It is also a basic string operation. It may cleave two strings or splice them in a specified way. Two examples are given below.

Rule 1: abcdef $\rightarrow$ abc, def

Rule 2: abcdef $\rightarrow$ abef, cd

Consistent with our principles thus far, "matter" must be preserved. That is, atomic objects are neither created nor destroyed by digestion, but rather they are rearranged. Rule 1 is a simple digestion that cleaves a string abcdef into two smaller strings abc and def. Rule 2 is a splicing operation that removes a substring cd from a longer string abcdef and leaves the remaining elements in a string abef. A more sophisticated digestion instruction to "remove" a substring from within a string might also be included.

COUNTING. Counting is not a simple operation in an ABC Machine. It must be accomplished as a distributed process. Let's look at two examples. The first is for a non-redundant system and the second one is based on statistical dataflow computing principles.

A simple non-redundant system may count as follows (note that there is more than one way to count). Suppose we wanted to count the number of atomic A's in Region 2 of the diagram above. The following rule leads to a final state with an indication of the number of A's:

CA^{n-1}, A $\rightarrow$ CA^{n}

The initial state is: {A, A, A, C}. After one execution of the rule, the state is {A, A, CA}. After three executions, the final state is {CAAA} or {CA^3} and the count is 3.

COMPARE. There are a number of ways to compare two strings. We may simply want to know which string is longer. In that case, the following rule does that where s1 and s2 are strings and t is an atomic tag and |s| represents the length of s.

$s1, s2, t \rightarrow ts1, s2$ if $|s1| < |s2|$, else $s1, ts2$

If we want to see if s1 is a substring of s2, the following rule will do that:

$s2 \rightarrow s1, x$

Here x is a variable string and the rule represents a splice type digestion rule with a variable string outcome. If the ABC Machine enforces conservation of matter, then this rule can only execute is s1 is a substring of s2.

MEMBRANE REMOVAL. As previously mentioned, regions (or membranes) may be created or destroyed. An instruction to remove a membrane may look like: X, E → X, E, Xmembrane, where "X" indicates which membrane. A similar instruction could be included to add a membrane, although this is more complicated because it must be specified where the membrane is to be added precisely.

ATOMIZER. AN atomizer instruction breaks a string into its component atoms. It is the ultimate digestion. I could be represented as:

$a_1a_2\ldots a_n, , E_{atomizer} \rightarrow a_1, a_2, \ldots a_n, , E_{atomizer}$

STRING MATCH. A more sophisticated version of COMPARE, the STRING MATCH instruction produces a goodness of fit between two strings, such as Hamming distance or some other metric of similarity.

ADDITIONAL INSTRUCTIONS. Additional instructions may be added (or deleted) as the ABC Machine architecture evolves in Phase 2..

Memory Characteristics.

Memory in the ABC machine is distributed across the regions and objects in the form of a content addressable memory. One could think of Short Term Memory (STM) as being the state of a region – that is what objects are present. This "state vector" represents a context for the region. All computing responds to this context or its STM. Hence, computing is driven by the data. An example to illustrate STM is the vector of string operands used to compute the "score" in the character recognition algorithm above. Regions representing characters that are chosen as the answer will have higher expression levels, or greater STM. This is a form of CAM, content addressable memory. Long Term Memory (LTM) is distributed across the entire ABC machine. It represents the rules or the program collectively that produce data. In particular, I/O instructions that operate on string operators and pass them to other regions embed this LTM. This is similar, but a generalization to, the weights in a neural network. If these instructions are modified, the way an ABC machine would "remember" a specific context would be entirely altered. An example to illustrate LTM using the character recognition example above is the set of instructions and their mapping to the processing regions – in other words the computer program itself.

Operating System Features.

An operating system is traditionally a software layer that mediates between an application program and the physical hardware. The operating system in the ABC architecture should in principle be the same. There has been no formal work to define or describe the operating system of a P System or membrane computer. This discussion is derived by direct analogy to biological cells by viewing them as computers.

However, there are some fundamental distinctions because of the distributed nature of the architecture and because of the possible inseparability of hardware and software in some implementations. The operating system of an ABC Machine is defined to mimic the properties of a cell operating system as necessary to support its definition. The remainder of this section defines and elaborates on the operating system of an ABC Machine.

Program execution in an ABC Machine is similar, but potentially far less complicated that transcription and translation in cells. A region in the ABC is required for program storage, which is called the nucleus for its analogy with real cells. The program storage consists of lists of possible instructions along with an "enzyme" or activating string for each. An operating system instruction must reside in the nucleus region that activates an instruction upon the presence of its enzyme by moving a "copy" of the rule associated with it outside the nucleus. Such an operating system instruction looks like this:
$I_{OS} = [O = \{ (\text{instruction-i, enzyme-i})\} \rightarrow P = \{ (\text{instruction-i} \mid \text{instruction-i})\}]$.
What this says is that whenever enzyme-I appears in the nucleus, put a copy of instruction-i in the main ABC Machine region and eliminate enzyme-i.

A cache memory of currently active instructions is analogous to mRNA in cells. Based on the definition of a ABC Machine, there is no absolute requirement for this as an operating system function. However, in many potential system designs, it may be desirable as a means of tracking activity. This could be accomplished by modifying the instruction above so as not to destroy the activating enzyme, but instead ligate it with a tag and save it. Then, <enzyme-I-tagged> becomes a measure of expression of instruction i. A means would be needed to extinguish the tagged enzymes after a period of time, such as a half life implemented by means of an appropriate supply of digestion instructions in the nucleus region.

As mentioned in the chart, there is no need for file management in a ABC Machine. Information is distributed, no history is maintained. The instruction set is a form of permanent memory. The expression levels (number of copies of each string) are short term memory and the implied regulatory networks based upon the active instructions and enzyme networks for activating other instructions (in other words the computer programs) are long term memory. Expression levels of strings store information. Regulatory processes controlling instructions manage this. The built in enzyme control of instruction activation is part of the operating system.

A mechanism for code optimization is not required by the posed definition of an ABC Machine. However, cells have one – mRNA splicing and splice variation. To the extent that a ABC Machine maintains a set of operating system instructions for alternative instruction compilation, it would exhibit similar functionality. This could be implemented through enzyme strings that control digestion instructions that splice out portions of other instructions. Because instructions are not strings, but instead relationships between strings, such instructions could only reside in the nucleus region and be controlled by the operating system.

Program execution requires several operating system support functions. One is that instructions must be delivered to the proper region. Another is that sometimes strings must be tagged for digestion to systematically eliminate them (corresponding to ubiquilation in cells). Finally, all strings and instructions must follow a Brownian motion (random walk) within their respective regions. The latter is a hardware (or shall we say wetware) function. The first two can be implemented by ligating tags to instructions and/or strings and having the existence in the operating system throughout the ABC machine of instructions to digest tagged strings and to tag them in the first place. The delivery of an instruction to an appropriate region in the first place should be encoded within the instruction string to begin with so that chaperone instructions will move them to the proper region when they randomly contact them. The result of all this machinery is the underlying mechanism of statistical dataflow computing. It is built into real cells, it can be easily simulated on a digital machine, but for all other ABC machine implementations, it must be crafted into the engineering of design. For example, let us suppose that instruction I_{KX} is the Kth instruction bound for region X. An operating system instruction to move it to region X is represented by:
$I_{KX} = [O = \{ (I_{KX})\} \rightarrow P = \{ (\mid |_X I_K)\}; E = \{\text{transportX-enzyme}\}]$.
The double vertical bar indicates that it must cross two membranes (nuclear, region X). the X subscript is moved from the instruction to the region to indicate the move.

Figure 12. Components of ABC Machine Operating System

Operating System Function	Biological Cell Behavior
Program Execution	Need region for storage area for programs (lists of available instructions and the enzymes that activate them). An activation instruction is needed as part of the OS that starts instructions when enzymes are there.
Cache Memory	A copy of each currently active instruction is maintained in the program storage region (nucleus). This is a desirable feature, but not required for a given ABC design.
File System Support	Files do not exist. Information is distributed and represented by expression levels of tokens, string and trees. There is no need for central support as it is handled intrinsically by the architecture.
Code Optimization	Mapping of instructions into variants based upon specific activating operands is a desirable feature of a ABC operating system, but not absolutely required.
Program Execution Policies	Signaling instructions to guide strings to their region of destination, tagging for deletion by ligation of a symbol, and doing these functions statistically is included.
I/O Control	Need operating system instructions to control endocytosis and exocytosis by forming temporary regions.
Memory Management	Expression levels of strings store information. Regulatory processes controlling instructions manage this. The built in enzyme control of instruction activation is part of the operating system.
Virtual Memory	Memory is not fixed. A ABC can continue to express many instructions and strings and is limited only by its total token material obtained through I/O operations and instruction execution.
Scheduling of Resources	There is no clock in a ABC. Program execution is parallel and distributed and operand driven. Strings such as enzymes schedule resources by their presence and hence the instructions that produce them and their activating enzymes are responsible for scheduling resources.

I/O instructions require operating system support. The biological process of diffusion can be permitted only if an inherent instruction designed to enable the passage of a specific string is built into the design baseline. Similarly, I/O instructions that mimic endocytosis and exocytosis require temporary regions that fuse with other regions to transport groups. The underlying mechanism to accomplish this should be supplied by the operating system.

In addition to the items mentioned in Figure 12, other housekeeping functions of real cells can be used as inspiration for the design of operating system functions. For example, in real cells, apoptosis means programmed cell death, or a biochemical pathway whereby cells destroy themselves when things go awry. A similar pathway may be needed in ABC Machines to eliminate regions that have failed as a means of error control. An example trigger for such a pathway may be the presence of a specific concentration level of a "toxic" string, <toxic-string>.

To recirculate atoms or strings for reuse (conservation of mass property), the atomizer instruction could be implemented in the outermost layer of the ABC Machine in high concentration. The operating system would then sprinkle "move" instructions throughout all membranes that randomly moves operators out one level at a certain rate. Eventually used and incomplete computations are recirculated.

It is possible to implement regions within an ABC machine that correspond to the "physiology" of organisms. The paragraph above implies a "circulatory system". Other regions of specializes functionality are possible (i.e., "central nervous system"), but the development of such concepts is left to a future task.

Another biological concept that may prove of value is the notion of molecular chaperones. In real cells, molecular chaperones are used to escort molecules to their proper organelles. In the ABC Machine, they may be thought of as internal I/O instructions of the operating system that move activated instructions from one region to another based on their encoded destination. A difference between such operating system chaperone instructions and normal program instructions is that they operate on instructions rather than strings and thus are the exclusive property of the operating system.

Cells have clearly understood functional organizations. Regions in cells are called organelles and each has a well known function. For example, the nucleus stores and processes the DNA and genes, the mitochondria produce energy from glucose and oxygen, lysosomes digest complex molecules, etc. A well designed ABC machine will have regions with specific functions. The need for a "nucleus" to store the computer program has already been discussed. An analog to the mitochondria is needed by any ABC machine that uses a concept similar to ATP or energy as an operand. It would act as the power supply, taking in external strings and converting them to energy tokens.

Lastly, the location of a membrane within an ABC machine must be an operating system function. In many systems, this may be merely random Brownian motion and in others this may be fixed coordinates as in cellular automata systems. But using biology as a model suggests that the most general systems must provide for partial, but constrained mobility.

1.8 Computational Analysis

Calude and Paun (2001) showed that P Systems with "enzymes" are computationally universal, that is Turing machine equivalent. Addison (2003) showed that the class of membrane computers that operate over string operators with enzyme-like operands, which subsumes the herein definition of the ABC Machine, is not only computationally universal (i.e., Turing machine equivalent), but that under certain configurations, machines in these classes may exhibit hypercomputing performance (i.e., exceed the Turing Machine limit). This would imply that the proposed ABC machine doing cognitive computing tasks could potentially exhibit hypercomputing performance for some configurations, but in the very least, it is Turing Machine equivalent. We believe, however, that the best use of the ABC Machine is in nondeterministic implementations of cognitive computing tasks with significant computational redundancy..

Turing machines are of theoretical interest because algorithms that can be computed on other architectures can be mapped into Turing machines. The well known thesis called the Church-Turing thesis that states that every "reasonable" computation or a Turing machine can carry out algorithm. This Church-Turing hypothesis implies that there is no need to create new computers to solve each type of problem -- if a problem would yield to a "reasonable" solution, a Turing Machine could solve it. According to the Church-Turing hypotheses, all modern digital computers are essentially Turing Machine equivalents. This way of thinking, however, ignores computational efficiency and economy. The word "reasonable" is extremely important.

A "reasonable" algorithm is generally considered to be on in which programs have a finite number of instructions from a finite set of possible instructions and which comes to a halt for valid input data that is when a solution has been found. The Church-Turing hypothesis says nothing about which procedures are considered to be "reasonable". Godel's theorem from mathematics proves that not every truth can be mechanically generated. Hence, there are problem classes that are not "reasonable". ABC Machines are a class of computer that brings economy and efficiency to certain problem classes that otherwise cannot achieve such efficiency on traditional computer architectures.

Since ABC Machines are P Systems (membrane computers) of a generalized form, then certain computing properties of P Systems may be extended to ABC Machines. Included in this section are key results of P Systems (Calude and Paun, 2001) that are then extended directly to ABC machines. The main result of studies of P Systems in membrane computing is that P Systems with catalysts (i.e., enzymes) are computationally universal (Paun2, 2002). This means that any problem that can be solved on a Turing Machine can also be solved on some P System with enzymes. It can then be concluded that ABC Machines are computationally universal or Turing Machine equivalent.

This means that any problem that can be solved on a Turing Machine, including all problems that can be solved on a von Neumann machine, can also be solved on some ABC implementation. This says nothing about whether the design and implementation provide an efficient or better solution to the problem, but nonetheless, the result is important.

Since nearly all work on P Systems to date has been based on P Systems with atomic objects only (i.e., no string objects) and has also been on P Systems with no statistical processing, the ABC Machine that enables a certain Turing Machine problem to be solved may be one that uses only atomic objects in single copies, although this is not necessarily the case. This implies that ABC Machines should be substantially more capable in computing than an ordinary P System. We have already shown that their statistical redundancy is an advantage for production systems with deep search spaces, and that pattern recognition programs have more instructional power than neural networks and can be programmed or optimized using "genetic programming" techniques.

The statistical nature of ABC Machines in particular is worthy of some note here. It stands to reason that if there exists an ABC Machine that can solve a problem with objects in singular copies (i.e., no statistical processing), then there exists a ABC Machine that uses the full statistical processing capability to generate an approximation to that same problem solution. The argument of proof for this conjecture is as follows. If a single instruction pathway can solve a problem that can be solved on a Turing Machine, essentially proved by isomorphism to Paun2 (2002), then a statistical implementation would have a large number of the same pathways. Not all of them would be executed at any point in time (i.e. have achieved a halting condition). The overall concentration of the resulting string objects represents an approximate solution.

What this means is that any problem that can be solved on a TM can be approximated on an ABC Machine (as well as solved). The approximate solution is of practical significance because large scale computing may require much less computational resources.

As described, it has long been assumed that the Turing Machine computes all "reasonable" computable functions. There have recently emerged a variety of papers claiming new computing models that have more than Turing Machine power. While such views are in the minority, it would be a mistake to dismiss them. Ord (2002) summarizes the work, including work with strong claims and support, such as Copeland (1998, 1999) and his coupled Turing Machines, Leeuw (1956) and probabilistic Turing Machines, and Spaan (1989) and her work on nondeterministic Turing Machines. Ord (2002) shows that each of these machine concepts has hypercomputing power, or beyond Turing Machine power.

Models of computation that compute more than Turing Machines are called hypercomputers or hypercomputing models (Ord, 2002). To show that a model of computation exhibits hypercomputing, one must show that it cannot be simulated on a Turing Machine, yet the new model of computation can simulate it.

Ord (2002) reports on a number of hypercomputing machines and examples. The present work shows that ABC Machines satisfy the definition of a number of the hypercomputing models summarized by Ord (2002) and others. Some of these concepts are purely theoretical and cannot be applied to ABC Machines. Copeland (1999) introduces the "coupled Turing Machine". This is a Turing Machine where one or more input channels provide input to the machine during the progress of computation (Ord, 2002). For example, one Turing Machine may input to another while it is in progress. A coupled Turing Machines differs from O-Machines and Turing Machines with initial inscriptions, in that they are finite (Ord, 2002), and therefore practical. Copeland (1999) showed that a coupled Turing machine could compute the halting function and all other recursively enumerable functions. Turing (1939) showed that some recursively enumerable functions required an O-Machine and could not be simulated by a Turing Machine if a halting function is required. Hence, the Coupled Turing Machine is a hypercomputing model. It is easy to see that an ABC Machine is a Coupled Turing Machine. Each membrane may be viewed as a single Turing Machine and since by definition, the membranes in an ABC Machine communicate with each other via I/O instructions, the ABC machine itself is a Couple Turing Machine in its most general form. Hence the ABC machine exhibits hypercomputing potential (depending on the selected instruction set, of course). This is a strong fundamental claim. An ABC machine is a finite machine by definition. Many, if not most, hypercomputing models proposed to date have been infinite machines not practically realizable.

Computational Properties of String Objects

As previously discussed, a string object is an object that consists of one or more atomic symbols. For example, the strong object "abc" consists of the atomic symbols a, b and c. As used in ABC machines, string objects may be modified by rules by appending additional atomic or string objects to them, or by ligating them into multiple objects. A rule operation over a string objects is deemed to take one computational step. There is no alteration to global memory requirements by string object operations because of the "conservation of matter" principle. In other words, any objects appended to a string must already exist before a rule is allowed to append the objects. When a ligation occurs, the resulting string objects may not be discarded unless a specific I/O instruction or rule causes that.

The computational burden of string object processing then lies entirely in the rule processing. A rule takes one computational step. The evaluation of its preconditions (enzymes, proximity, etc.) takes an additional amount of time. The true computational time can only be assessed by considering the fact that there may be a large number of copies of the same rule and operands at the same time in an ABC Machine. If there are N copies of a string object "abc" that is operated on by a rule or instruction R that operates in the presence of enzyme E and there is an adequate supply of E, then the computational time for executing R must be somewhere between 1 and N time steps. It will take 1 time step if all N copies are operated on by a copy of instruction R at the same time and it will take N steps if they are all operated upon sequentially. Neither of these cases is realistic, it would be expected that this operation would occur in a statistical fashion similar to a biochemical reaction process.

To quantify this, consider a more precise statement of an example concatenation instruction R as follows:

R: E, a, bc → E, abc

To borrow from the field of biochemistry, the symbols [E], [abc], [a] and [bc] shall be interpreted as "the concentration of" E, abc, a, and bc respectively (Becker, 2000). In biochemistry, this means molar concentration. However, the convention will be adopted that [x] means the concentration of x in units of 1000 copies per unit area where a unit area is the average membrane size. Hence, if [x] is 4, then there are 4000 copies of the operator x in the membrane under discussion.

Having adopted this convention, it is now possible to further borrow from the principles of reaction kinetics in biochemistry to assess the rate at which a computation over a string object will occur (Becker, 2000), (Bower and Bolouri, 2001). Recall from the discussion above that an instruction R over a string object takes one computation step. The time associated with that step is dependent upon several factors including the nature of the "hardware" implementation and its components. Borrowing from the world of biochemistry again, this is normally captured as a rate constant. Hence, the instruction R may be represented as follows, capturing the rate constant and the concentration levels of each string object, where k is the rate constant of the instruction:

$$R:\quad a, bc \xrightarrow{k} abc$$

Since an instruction execution is completely isomorphic to biochemical reactions, the reaction kinetics principles (Bower and Bolouri, 2001) may be used to derive the rate equations of a ABC instruction. The instruction R proceeds then as follows:

$$d[a]/dt = -k\,[a]\,[bc]$$
$$d[bc]/dt = -k\,[a]\,[bc]$$
$$d[abc]/dt = k\,[a]\,[bc]$$

These are differential equations in state variables that are equal to the concentration levels of each object. Such differential equations can then directly model the "state".

The statistical nature of the processing (i.e. a large number of copies of operands and instructions) cannot be separated from the computational performance of instructions over string operators. The analysis above resulted in the description of the structure of a differential equations model for rate kinetics associated with computational instruction processing. The instruction execution rate, and thus the computational time of a statistical computational process is predicted by the solution of a simultaneous system of nonlinear differential equations as previously described.

There are circumstances when the use of differential equations to analyze processing time is overkill. However, when the computational process is statistical, this is a reasonable model. If there were only a single copy or a few copies of each instruction and operand, then the differential equations would not be a good approximation to the process. Under such circumstances, with a fixed interval of time (for a single copy) or a stochastic state transition model would make more sense. But since the definition of an ABC machine called for statistical processing, or a large number of copies, the differential equations model for computational time progression of an instruction shall be considered the best model.

1.9 Programming Concepts

An ABC Machine may be programmed manually or by using genetic algorithms. By programming, it is meant that the ABC Machine is configured in a manner to solve a specific problem. This means that a set of tokens, rules and regions must be selected along with an initial condition, in such a manner that their execution will cause the problem of interest to be solved.

An ABC program is fully specified by the following there items:

- Region map
- Instructional set and number of copies in each region
- Atomic elements and strings and their initial distribution

This is a simple, but not explicit program. It is simple in the sense that once specified, the program, "runs" by repeated application of the same set of procedures. Therefore, ***software engineering should result in relatively few bugs compared with conventional computers***. The fact that it is not explicit makes the programming as much an art as it is a science. Trial and error will be required. However, for well structure large scale problems, genetic programming may be used to optimize the program. This concept was developed analytically during Phase 1 and will be implemented in Phase 2 as part of the SOW.

This rest of section describes the concept of genetic programming as a means of programming an ABC Machine. The Phase 2 SBIR proposal will implement this as a method for programming the ABC Machine for large scale well defined problems such as face recognition. Genetic programming is already a developed area of Computer Science. An ABC Machine has a highly nonlinear space that cannot be fully pre-stated. To program it is very similar to solving a complex nonlinear optimization problem in a multi-dimensional space.

Programming an ABC Machine using a genetic algorithm requires beginning with a random population of ABC Machine configurations that appear to solve the problem. Each of the members (a member is a set of instructions and operands arranged in a set of regions to solve the problem) of the population will be tested against a fitness function as the genetic programming algorithm is executed. The fitness function must be chosen as a desirable outcome and a distance metric to it in terms of the state vector of the ABC Machine. Only the relevant output parameters are used in this assessment.

Random variations are made to the variable parameters in the ABC Machines. These random variations include changing operands, enzymes or products in instructions, or changing the membrane into which a particular instruction or pathway is assigned. While the algorithm specifies that these variations are to be random, intuition suggests that there is room here for making intelligent choices to accelerate the programming. This can only be tested empirically. This is obviously a large and nonlinear search space and can only be solved empirically such as with the proposed genetic programming technique. Mutation, reproduction and crossover are simply redefinitions of the ABC Machine configuration based on the shuffling of parameters so as to provide new populations for testing on the next iteration of the algorithm. A specific example of this concept will be programmed and tested during Phase 1.

First, some examples of parameters that can be controlled for an ABC Machine are described. The instruction is the basic unit of the ABC Machine program. Parameters of an instruction include the operands, products and catalysts (tokens or strings which must be present in order for an instruction to execute). Changing the strings that make up one or more products, catalysts and/or products in such a way that the instruction remains well formed may parametrically modify instructions. By a well formed instruction, it is meant that tokens making up strings are neither created nor destroyed by execution of the instruction. Changing the regions to which products are delivered, but this is a more drastic modification may modify I/O instructions. For use of genetic programming to alter an ABC Machine program, the present work restricts the parametric changes to changing the strings that make up the products, catalysts and operands in a well formed way.

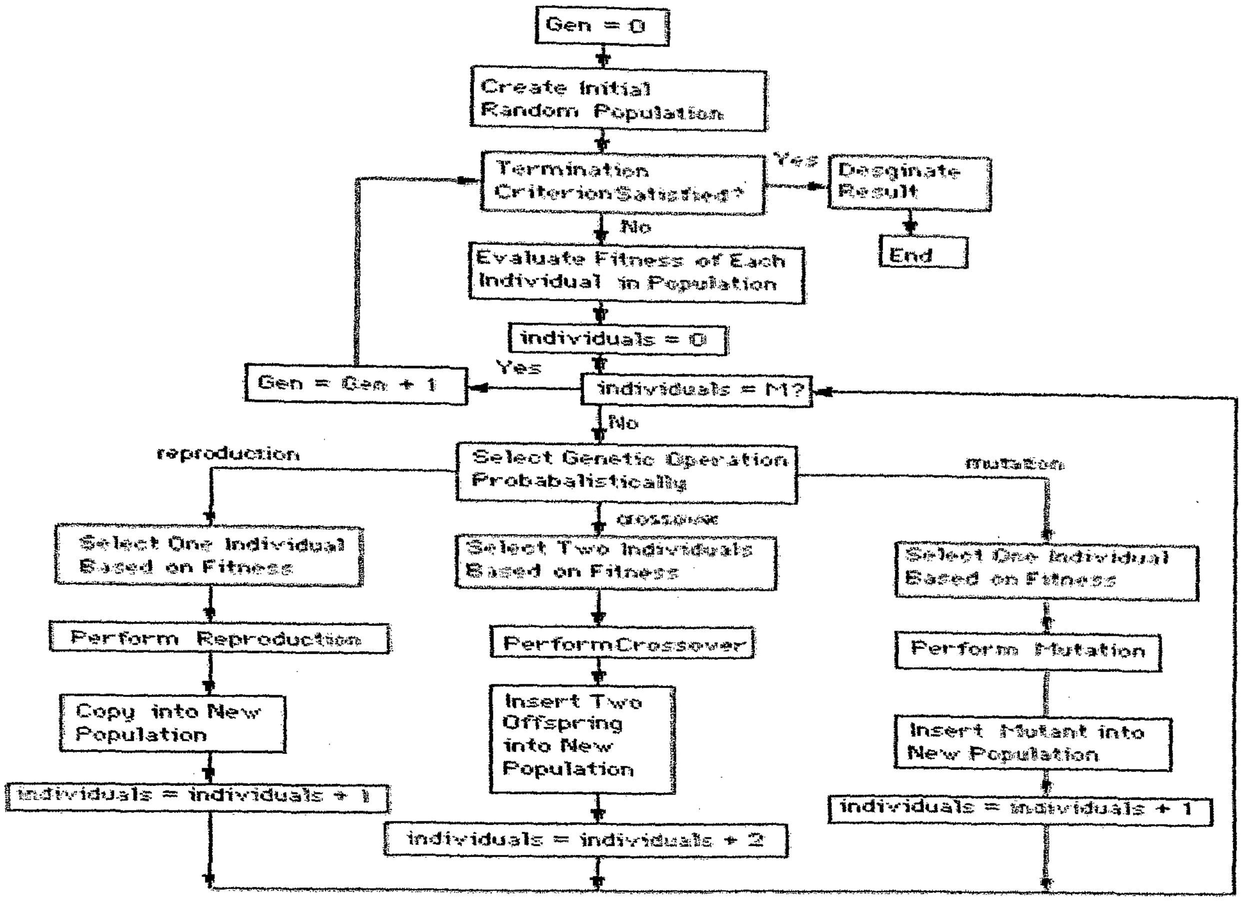
There are two levels of programming that must be accounted for. These are the programming of an individual ABC Machine rules and tokens through genetic recombination. It must be accomplished by using genetic recombination to select parameters for individual ABC Machine configurations (software).

It is also important to stress that ABC Machines may be implemented manually rather than through genetic programming. This is an engineering decision. It may be easy or convenient to set up an initial solution. But due to their high complexity, it is unlikely that a manually derived solution will be optimal in any sense. Further, it is difficult to imagine that traditional analytical techniques can be used to derive an optimal design in a problem space with such high state dimensionality by other than an empirical optimization method like genetic

As an example to illustrate programming the ABC machine by genetic programming, consider the character recognition program previously discussed. Suppose the implementation called for one output region for each character and an expression vector corresponding to a fixed set of strings to count the output. One may represent the instruction set in the regions parametrically – using a random collection of ligation, digestion, logic and I/O operators that could be "genetically recombined" by exchanging pieces of the instruction with other instructions. For example, the two instructions:

Figure 13.

Flowchart for Genetic Programming


Gen = 0
Create Initial Random Population
Termination Criterion Satisfied?
Yes
Desginate Result
End
No
Evaluate Fitness of Each Individual in Population
individuals = 0
Yes
Gen = Gen + 1
individuals = M?
No
reproduction
Select Genetic Operation Probabalistically
mutation
crossover
Select One Individual Based on Fitness
Select Two Individuals Based on Fitness
Select One Individual Based on Fitness
Perform Reproduction
Perform Crossover
Perform Mutation
Copy into New Population
Insert Two Offspring into New Population
Insert Mutant into New Population
individuals = individuals + 1
individuals = individuals + 2
individuals = individuals + 1

First, some examples of parameters are described. The instruction is the basic unit of the program. Parameters of an instruction include the operands, products and enzymes. Changing the strings that make up one or more products or enzymes and/or products in such a way that the instruction remains well formed may parametrically modify instructions. By a well formed instruction, it is meant that tokens making up strings are neither created nor destroyed by execution of the instruction. Changing the regions to which products are delivered, but this is a more drastic modification may modify I/O instructions. For use of genetic programming to ~~alter a~~ program, the present work restricts the parametric changes to changing the strings that make up the products, enzymes and operands in a well formed way.

There are two levels of programming that must be accounted for. These are the programming of an individual ABC machine through genetic recombination. It must be accomplished by using genetic recombination to select parameters. The latter may require that not all machines are programmed the same, and therefore some degree of genetic recombination akin to developmental biology and its associated genetics may be required. While this may sound extremely complex, and it can be, the approach taken here is a simple first order implementation plan of each.

It is also important to stress that programming ABC machines may be implemented manually rather than through genetic programming. This is an engineering decision. It may be easy or convenient to set up an initial solution. But due to the high complexity, it is unlikely that a manually derived solution will be optimal in any sense. Further, it is difficult to imagine that traditional analytical techniques can be used to derive an optimal design in a problem space with such high state dimensionality as an ABC machine by other than an empirical optimization method like genetic programming.

Programming requires beginning with a random population that appear to solve the problem. Each of the members of the population will be tested against a fitness function as the genetic programming algorithm. The fitness function must be chosen as a desirable outcome and a distance metric to it in terms of the machine state vector. Only the relevant output parameters are used in this assessment.

Random variations are made to the variable parameters, as the case may be. These random variations for an ABC machine include changing operands, enzymes or products in instructions, or changing the membrane into which a particular instruction or pathway is assigned. While the algorithm specifies that these variations are to be random, intuition suggests that there is room here for making intelligent choices to accelerate the programming. This can only be tested empirically. This is obviously a large and nonlinear search space and can only be solved empirically such as with the proposed genetic programming technique. Mutation, reproduction and crossover are simply redefinitions based on the shuffling of parameters so as to provide new populations for testing on the next iteration of the algorithm.

Figure 14. Genetic Programming of ABC Machines

Genetic Programming	Application to ABC
Initial Random Population	Initial Instruction Set and Membrane Mapping for N variations
Termination Condition	Fitness Function of Best Run Exceeds Preset Threshold
Fitness Function	A distance metric is applied to relevant subset of the state vector
Mutation	Randomly vary an instruction parameter or membrane boundary
Crossover	Combine half the instructions and membrane boundaries from one ABC with another
Reproduction	Define a new ABC by combining the crossover information above

1.10 Cluster Mapping

Based on a review of the commercial opportunities, an evaluation of potential physical implementations, and attendance at the July 2005 workshop on Cognitive Computing at Cornell University, it was decided that the best near term implementation of the ABC Machine architecture is on a high performance cluster (HPC). There are several reasons for this conclusion:

- The market for specialized computer hardware boards is fragmented
- The Cornell workshop identified a need for more compute power applied to Ai and cognitive computing algorithms, rather than customer machines, essentially pointing toward using high performance clusters to emulate the ABC Machine rather than customize them
- While there is a proliferation of HPCs in the market, the software for them and utilization of them at full capacity is lagging behind, leaving the door open for more applications
- Biological implementations are not feasible in the immediate future, although promising for the far future
- Single machine emulation will not offer enough compute power for the redundancy required by the ABC Machine architecture

This section describes how the ABC machine or Agent Based Computer can be mapped onto a HPC as determined by our Phase 1 study. In review, an "ABC Machine" is a biologically inspired computing device that works on complex problems by defining those problem in terms of massively redundant strings and operations on those strings over time. Solutions are presented in terms of the state of the machine at a given point in time in the future and vary according to the way the problem is defined.

We proposed implementing the ABC machine on a cluster because of the wide availability of existing software to program one and the ease at which the size of the cluster can be scaled. Clusters are cheap to build and widely available using commodity hardware. Developing the implementation of the ABC machine with libraries like MPI ensures the software will be portable, easier to maintain, and extensible.

The following are the key features available when implanting an ABC Machine on a HPC:

- Threaded message passing and Core operations for maximum CPU use.
- Minimized administrative message-passing overhead helps to scale processing power relatively with the size of the cluster
- Aggregate memory and CPU speed allows solution-seeking over a large problem domain
- Standard cluster programming library MPI used to encourage portability, maintenance, and extensibility
- The machine will be straight-forward to program for a knowledgeable researcher/user with reasonable programming skills.
- Interfaces built on modern programming standards and APIs facilitates incorporation of the ABC Engine as a component in larger software projects
- Includes an optimized set of Core Foundation Rules that can be extended and/or grouped together to form more powerful string operations (ie Enzymes)
- User-adjustable Atomic Density and Rule Density to simulate organic concentration
- The current ABC Engine is written in C for speed and final implementation is proposed to be written in Object-Oriented C++ for even greater encapsulation, abstraction, and flexibility.

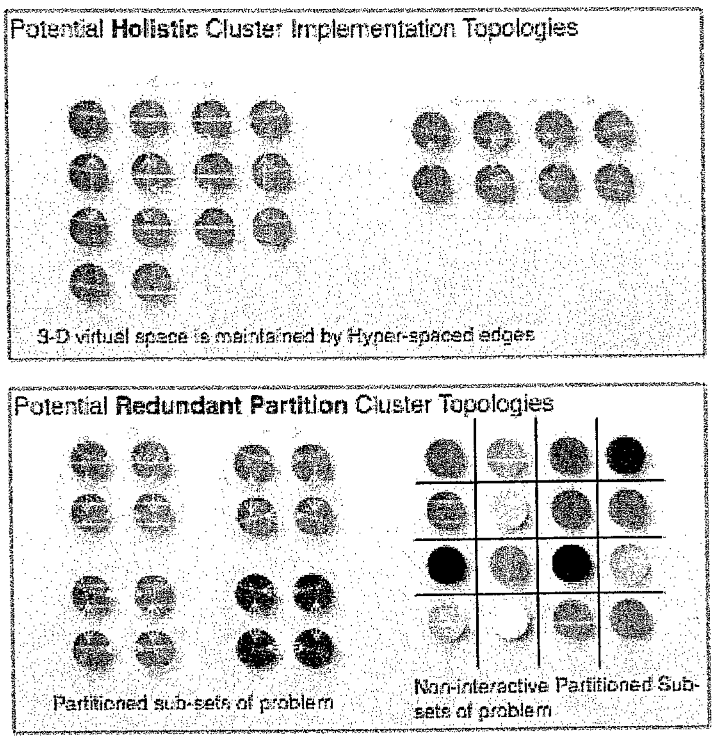
There are two contrasting approaches for mapping the computations from the ABC architecture onto an HPC. We call these "holistic", where the entire cluster acts as a single device, and "redundant partitions", an approach which takes advantage of MPI (message passing interface). Each is described below:

Holistic Approach. In the Holistic approach, the entire cluster acts as a single device and every atomic element exists in a single Universe. Because there is only one Universe, or problem space, the number of individual atomic elements can be very large. As each node on the cluster acts on a sector of the entire space, the size of the problem space is limited only by the aggregate size of the cluster's memory. Additionally, a large atomic element size preserves the statistical nature of the ABC machine's operation and makes generalizing the solution more straight-forward.

The Universe is mapped into *n* sectors, a sector being an MPI node, with open boarders. Atoms are free to travel between the sectors of the Universe and beyond the hyper-space edges. In other words, they wrap-around so that atomic elements are not reflected at the Universe's virtual boarders. Initial plans for the topology included requiring rigid geometric sizes (like perfect squares and/or perfect cubes) but have since been dismissed. Instead, more focus is dedicated to distributing data and establishing the initial state of the machine.

The initial region structure is defined at the start and remains fixed. In the Holistic approach Regional changes are not allowed. Because of the asynchronous nature of regional construction and destruction, it would be necessary to centrally define the data structure to represent the regional space. Updating the data structure would require momentarily suspending all operations throughout the cluster to preserve regional integrity of the system. Simultaneous regional operations across the cluster would result in massive network overhead and frequent suspension of the system preventing real work from being accomplished. The problem would get worse as the size of the cluster increased: if the nature of the problem necessitated a large number regional changes the cluster would drown in regional maintenance overhead.

Figure 15
Cluster Mapping


Potential Holistic Cluster Implementation Topologies
3-D virtual space is maintained by Hyper-spaced edges
Potential Redundant Partition Cluster Topologies
Partitioned sub-sets of problem
Non-interactive Partitioned Sub-sets of problem

Redundant Partitions. Redundant partitioning involves a more traditional MPI approach to implementing the ABC machine. Instead of treating the cluster as a single space, problems are initialized and then distributed as copies to individual nodes or to groups of nodes. Over *n* nodes, the user has the prerogative to duplicate the problem *n* times or to partition the problem into subsets.

Once the nodes begin their work, individual atoms remain in their space and are **not** passed to other nodes. The initial region structure is defined at the start but can be changed by local execution of Rules. Because of the constricted nature of the partitions, operations requiring regional changes can be allowed for algorithms that require them resulting in lower network and CPU cost. More research needs to be done to determine if allowing regional changes on small, limited-sized cluster groups would be a reasonable compromise to the Holistic approach. By keeping the sub-groups/limited-sized clusters small, the increased regional maintenance overhead is allowed to permit operating on larger partitioned problem sets.

Data Representation. In the ABC machine, data is represented by strings of atoms. An atom string can be one atom long or many atoms long. This cluster implementation uses null-terminated strings of unlimited size to represent both individual and chains of atomic elements. The regional location of the atom string is stored in the first five characters of the string. For example, an atom located in Region One with an atomic string consisting of "aaabbbcd" is represented as "00001aaabbbcd".

There is a core set of instructions that deal with atom strings on a basic level. Core instructions include creating and destroying atoms, concatenating and digesting atoms, rotating elemental strings in the atom one character left or right, and removing the left-most or right-most elemental character from the string. Additional core instructions provide for determining the magnitude of atomic strings and for comparing different strings. Other instruction can determine regional location of an atom string and which region encompasses an atom's current region. More core instructions can be added. There is also a Core set of instruction for promoting/demoting atoms to other regions. The idea behind having the Core set of instructions is that programming the ABC machine is easier with such a foundation and makes building more complicated 'enzymes' or Rules easier. We parallel this idea with basic instruction sets for a CPU upon which complication Operating Systems and GUI are built.

A Universe is represented as a global array of pointers to arrays of Atoms. Using a global datastructure allows access to the regional construct of the Universe without needing accessor functions. That is important as Rules become more complicated and nested because a series of Rules can create a set of new atoms that do not then all exist in the same region. Also, we have included the idea of Density in both Atomic and Rule content: more densely-occurring elements should have more frequent reactions.

The Shuffler. The "shuffler" is the algorithm used to simulate motion in an ABC machine. By motion, it is meant that the operands move around until they come into contact with instructions in dataflow like fashion. Unless a biological implementation is used, such motion is only a concept and must be "simulated". The central operation of the Shuffler in the Universe works like this:

- A region is selected at random depending on the atomic density of each region. It makes sense to weigh reactions to denser atomic content because the more atoms there are in a location the more often there would be an interaction between them.
- A Rule is then selected from a table of Rules that exist within that region based on the density of Rules for that region. Obviously, if there are 100x more 'comparison' Rules then there are 'rotation' Rules in a given region the system should reflect that in selecting how the atoms interact. Currently, Rules are defined with a single interface type that takes a pointer to all of the atoms in a particular region and the Rule should return a temporary structure that contains all of the newly created and/or changed atoms. By implementing this common interface, new Rules can then be built from combinations of existing Rules and dealt with in a consistent way.
- The Rule dictates how many atoms should be randomly drawn from the local region. The selected Atoms are then processed by the Rule.
- After the Rule is finished, it returns a list of all of the Atoms created during processing to the Shuffler. The Shuffler then has the responsibility to place the Atoms into their correct Region.
- The process repeats. Note that a Rule is allowed to request data from the system and to request that the Machine should terminate.

Atomic Character Representation. Because characters are limited to an 8-bit representation, the current prototype of the ABC machine has a symbol alphabet limit of 250. With 256 possible 8-bit combinations, the 250 character limit reserves a terminating, or null character, and four special characters for system use. UNICODE would be a preferred minimal atomic representation because of the raised limit of 65,536 unique symbols. Ultimately, an object-oriented approach (in C++ perhaps) would yield an encapsulated Atom object with no meaningful limit on unique symbol labels.

Input/Output and Programming. MPI is a common, robust framework used to develop software solutions on most current Beowulf cluster implementations. Therefore it is convenient that the "ABC Machine" utilize these libraries for communication. In both approaches, the "ABC Machine" is written and launched as an MPI program. and in both approaches the MPI libraries abstract the bulk of transmitting the initial data needed for computation, and (in the holistic approach) passing atoms between nodes in the system. MPI is also key in system management of the overall system, such as stopping/pausing operation, initializing data, and collecting results.

Even though prototyping the machine is set up using hard-coded problems, Phase II design involves configuring and launching the system by a specifically-defined XML file. The file would contain directives such as implementation approach, initial atomic content, density of atoms, Rule definition and densities, regional structure, status queries, termination time, additional files, etc. Defining the problem set in XML lends the power and portability of the XML standard and a rich set of software designed to parse, produce, and manipulate XML. (eg. Text Editors, Web/Grid-Services, GUI Front-ends, Apache Xerces parser, etc). Future development could feature a parsed scripting language to define and distribute the problem. Every stage of development has been driven by the goal of open-ended scalability and transparency to the problem designer. Output can be delivered in a variety of formats. These include plain text logging, XML/XHTML, and potentially Relational Databases (MySQL, Postgresql) for collection of cluster samples/snapshots over intervals of time.

An prototypical example of an XML file to configure and launch the ABC Machine:

```
<abcSystem targetnamespace=>"http://lexxle.com/abcMachine">
<!--
mode values are holistic|partitioned
-->
<init mode="holistic">
<!--
specifies how long to allow system run before terminating it

Time Modes:
        seconds:        Length of timer interval in Seconds. 60 is one minute
        iterations: Number of times the timer will restart or negative for continuous.
-->
<timer seconds="60" iterations="-1"/>
<!--
Initial Region Setup
-->
<region id="0">
        <!-- initial atoms in this region -->
        <!-- set-up wildcard character -->
        <wildcard value = "a..z"/> <!-- All characters a to z inclusive -->
        <atom count="1000">abbacd</atom>
        <atom count="1000">gefjig</atom>
                <!-- randomly generate atoms with 4 wildcard
                chars, then 4 b's then 5 wildcard chars -->
        <atom count="5000">.4?.bbbb.5?.</atom>
        <!-- enzyme name will be bound to a Rule (possibly scripted) -->
        <enzyme name="PathDiscovery"/>
        <!-- an Inner Region initially with no atomic elements -->
        <region id="1">
                <enzyme name="PathRefinement"/>
        </region>
</region>
</init>
</abcSystem>
```

Future Implementation Concepts. Best efforts will be used in Phase 2 to accomplish the following enhancements. Due to the speed and support of MPI for C/C++ on Beowulf clusters, these languages could be ideal choices for further development the "ABC Machine." Future implementations however, may span beyond just that of a cluster as other cutting edge technologies such as JXTA are evolving and may offer an efficient way to run an ABC Machine in a more peer-to-peer environment similar to that of the infamous "SETI at Home" project.

By using the XML standard, the ABC machine could be included as a workflow in a Grid Computing platform or could be used as a logical first or next step in a series of intelligently-designed workflows.

The following concepts are further open issues that will continue to be explored and implemented on a best efforts basis in Phase 2.

- Machine Permeability and how the engine deals with 'slow' nodes. We currently treat the cluster as a system of similarly-built computers.
- Initial Network Cluster evaluation per node and how it relates to distribution of data
- Concurrency scheduling and/or pipelining for increased per node performance
- Complexity Calculations
- Specifically evaluating the state of the machine ie solutions.

1.11 Phase 1 Results

In Phase 1, our effort not only defined the ABC architecture in detail, but also developed the mapping strategy to a cluster machine. Some preliminary results have been obtained by comparing the use of the ABC Machine to a neural network for character recognition.

The ABC Machine VS Artificial Neural Network

Character Recognition trials were done comparing classification ability of the ABC Machine versus a biased, hidden-layer, feed-forward network consisting of 25 inputs, 50 hidden-layer nodes, and 12 outputs. The 25 inputs each corresponded to a 'grid square' of the input character being classified and the twelve outputs were a set of switches with a value of '1.0' in the corresponding output and '0' in all of the other outputs. Training was done over 250 epochs of each set of 12 input/output pairs with a randomized initial set of initial weights and sigmoidal outputs. Classes consisted of 12 characters 1, 2, 3, 4, 5, 6, A, B, C, D, E, & F. After training and with no mutation, the ANN could accurately classify input 100% correctly with no errors.

The ABC machine was programmed to classify inputs based on Hamming distance only where the Hamming distance is defined as the total number of non-matching bits between the subject and the class.

To test inputs, mutations were performed by randomly alternating a percentage of the representation bits, or "flipping them," where '1' would be changed to a '0' and a '0' would be changed to a '1.' Four different mutation percentage settings were used: 0%, 5%, 10%,15%, 20%, 25%, 30%, 50%, 100% to simulate noisy input signals. Percent mutated was the probability that a given grid sector got flipped in error during detection.

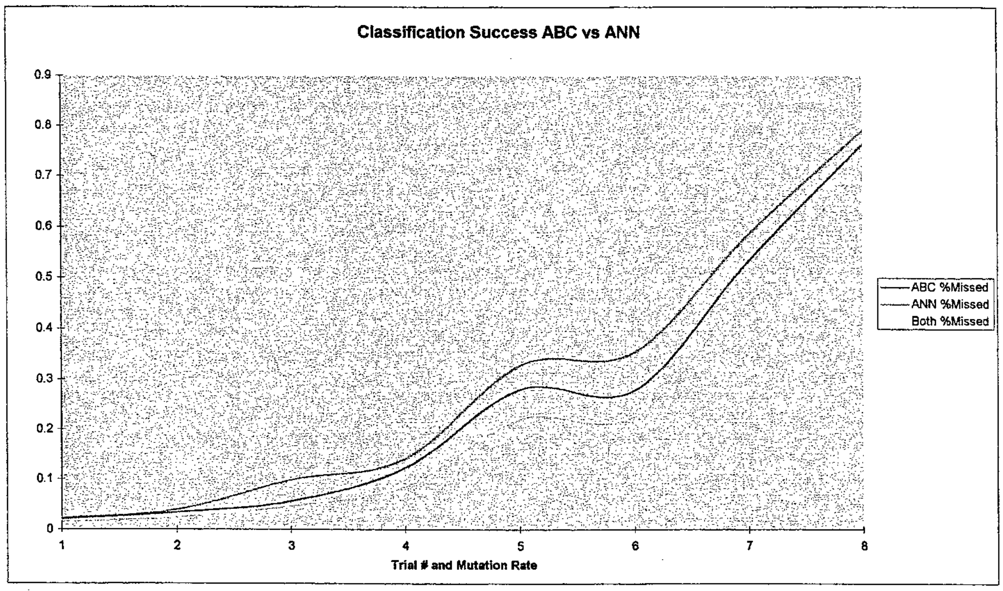


Figure 16
ABC vs. ANN

Each classifier was presented with the same input signal and their computed outputs were compared. After each trial set, the ANN was retrained on a re-randomized set of weights to minimize the likelihood that the ANN got trapped in a 'local min' that reduced its ability to classify a particular input well.

After a series of trials it was discovered that the simplified ABC string matching rule set performed as well as the ANN over 'perfect' input data, as was expected, but also significantly better - up to 50% better in some cases - than the ANN over progressively mutated inputs.

To judge the quality of classification, not only were each classifier's misses accumulated but also a record was kept when they *both* miss-classified so that there was an indication of how mutated a subject's representation data had become. Obviously, using random mutation leaves open the possibility that the input data will be so obscured as to not be able to be classified by any means.

The most promising aspect of these initial results are that the classification used by the ABC machine is an extremely simple comparison method.

The ABC Machine vs. Classical Single CPU Symbolic Computing

A symbolic computing program on a single CPU machine may take a long time to search and may be brittle at the boundaries if rules do not exist to continue the search strategy. A symbolic computing problem implemented on an ABC Machine architecture degrades gracefully because of the many copies of the operands and instructions and the multiple redundant paths being pursued. Further, it can search deep search spaces due to the inherent massive parallelism. The shortest path algorithm explored and the methods to augment it for best first search illustrated a way to implement symbolic computing problems on an ABC Machine architecture.

It is already known that "best first or heuristic search can greatly reduce a search space. Consider, for example, a very large search space such as the game of chess. It may have some 10**20 possible search nodes if brute force is used. If a heuristic can reduce the decisions at each step by 90% or more, then there may be a 10**3 to 10**5 improvement in search, reducing the total search nodes to say 10**16, still a very large number. Now consider an ABC Machine architecture on an 1100 dual CPU cluster such as System X at Virginia Tech. One could do say 10**4 "rats in a maze" approach and reduce the search space to say 10**12 single time steps which is getting much closer to a computable number. But this achievement is made only due to rundancy and sheer parallelism. Such solutions will be much more dramatic in the future when either biological implementations of ABC allow, say 10**6 nodes, or else clusters get bigger and faster, which they inevitably will.

But the above argument leaves out one significant advance of the ABC machine architecture. That is the ability to encapsulate rules in membranes and thus restrict the amount of processing done locally. This has the net effect of further improving the heuristics if it is "programmed" with a genetic algorithm as proposed above. Hence, another order of magnitude advance is possible with the HPC implementation. The hypothetical chess problem discussed above could be reduced to say 10**9 steps, now a very doable number on an HPC. Hence, we argue that there is not only a parallel processing advance, but a heuristic power advance due to the combination of genetic programming and local rule sets in the ABC architecture. The Phase 2 program will prototype this completely and seek to prove this significant result.

What makes the ABC Machine a strong and viable architecture?

The ABC Machine can be implemented today on cluster machines, rendering a high compute power alternative to traditional cognitive computing or AI.

It has been shown to be capable of large scale pattern recognition problems – that is, problems with a large number of classes, no need for training, and ability to recognize in the presence of noisy data.

The ABC Machine is capable of easily trading off compute time for more space, enabling NP complete problems to be solved on polynomial time.

It an be used for large scale heuristic search problems with good results and graceful degradation. It can also be used for complex systems simulation.

It is very important to understand that the power of the ABC machine is in its extensibility. The framework of the Universe and the method by which to create and connect regions has already been established. The only thing the user has to do is design the regional construction, create a profile of the initial atomic content of each region or the universe as a whole, and then define a rule set.

The rule set is the foundation of how the ABC Machine solves problems and the code is fully Object-Oriented. To program the machine, the programmer currently only has to implement a single Interface, define those three qualities, and then let it go. Because the complexity is abstracted from the user/programmer, ABC Machine programs are easy to create, install, maintain, and extend. While there are plans to create advanced high-level tools incorporating GUI and scripting options, the ability to actually 'get into the machine' will always be a possibility for the power user.

4. Related Work.

4.2 Prior Research

This SBIR project is based on an extension to the doctoral dissertation of the PI, Dr. Edwin R. Addison. As indicated in his bio, Dr. Addison is an accomplished entrepreneur with 2 previous SBIR funded companies with successful exits, 20 years of part time teaching experience in Computer Science (including artificial intelligence) and 25 years of professional experience. He completed is doctoral work in the area of biologically inspired computer architectures recently (2003). His doctoral dissertation representing several years of work was: "The Whole Cell Computer: A New Computer Architecture Based on a Cell Biology Metaphor", Edwin R. Addison, 2003. In this work, he defined a derivative of membrane computers to perform symbolic computation. An extension of the concepts developed there are modified and proposed for this SBIR topic.

4.3 Relevant Experience of the Lexxle Staff

The following are the most significant qualifications of Lexxle, Inc. for this work:

- Ed Addison – founder and CEO with two former successful search ventures. Dr. Addison has both technical and business/management skills. His bio is below. He has managed several early stage ventures, numerous Government contracts and several commercial software products in the search area. He is also an adjunct professor with Johns Hopkins University.

- Ms. Terri Hobbs is a engineer on the project and lead developer. She has extensive experience with natural language processing and with search engine software (see bio below).

- Lawrence Husick, founding CTO of Infonautics (an information systems venture that went public in 1996), is also acting as advisor to Lexxle, inc. He will advise on commercial product specifications.

- Lexxle, Inc. has employed well qualified software engineers, including Dr. Kathleen Romanik and Ms. Chris Eck are both experienced in writing code for search applications (bios below). Lexxle, Inc. has a relationship with University of North Carolina where graduate students are available for part time employment. The company also has several additional software engineers and several in the hiring pipeline who may be applicable.

- Lexxle team specific project experience. The company has experience across a range of projects that establish qualifications for this work. Some of the more significant experiences are cited below:

  - Powerize.com – Dr. Edwin R. Addison and Kathleen Romanik and Ms. Terri Hobbs of Lexxle, Inc. all were former founding members of Powerize.com, an Internet business information search engine that was merged with Hoovers/Dunn & Bradstreeet in 2000. While at Powerize.com, technical work using the IBM (formerly Booz, Allen and formerly ADS) Minerva platform formed the basis of the Powerize.com search engine. In addition, extensive "hidden web" databases were searched using a federated search strategy and presented on the web site. Powerize.com represents a state-of-the-art search experience for the Lexxle team. Powerize.com was partly funded by Air Force SBIR contracts.

- BioSequent, LLC. The same team also worked for start up BioSequent in 2002, a biotech search engine company that is no longer funded. While at BioSequent, technical work involved natural language extraction (identifying protein names within research journal articles) as well as BLAST search, and integrating search results from multiple databases using an evidential reasoning algorithm. BioSequent did not obtain continued funding, but the remaining work has become part of Lexxle.

- ConQuest and DARPA. Dr. Addison and Ms. Hobbs of Lexxle, Inc. led a DARPA Phase 2 SBIR to integrate Word-Net into the ConQuest search engine in 1994-5. ConQuest is a search engine based on a semantic network that is now Convera's RetrievalWare.

- ConQuest and UMLS. Dr. Addison and Ms. Hobbs led an NIH Phase 2 SBIR to integrate the UMLS language system form NIH into the ConQuest search engine in 1993.

- Extensive additional customer experience through ConQuest including the Air Force, DOD, Intelligence Community, publishers of information, online services and Fortune 500 companies.

- Natural Language Processing – Dr. Addison taught Natural Language Processing for Johns Hopkins University form 1986 until 1995. He continues to teach courses in Computer Science at Johns Hopkins University on a part time basis.

■ Multiple Commercial Venture Experiences. The team behind Lexxle has successfully started, developed and merged several previous start up companies over the last 15-20 years. This experience and track record represents a greater probability of success and a reduced risk for Lexxle, Inc.

4.4 REFERENCES

Addison "The Whole Cell Computer: A New Computer Architecture Based on a Cell Biology Metaphor", doctoral dissertation of Edwin R. Addison, 2003

Anderson, J. "The Architecture of Cognition", Cambridge, MA: Harvard University Press, 1983.

Bar-Yam, Y. (1993) Dynamics of Complex Systems. Reading, MA: Perseus Books.

BICA, "Biologically Inspired Computer Architectures", DARPA Research Program, 2005

Bower and Bolouri, Computational Modeling of Genetic and Biochemical Networks, 2001, McGraw Hill

Calude, C., and Paun, G. (2001). Computing with Cells and Atoms: An Introduction to Quantum, DNA and Membrane Computing, Chapter 3. London: Taylor & Frances.

Charniak, E. (1986). Principles of Artificial Intelligence, Chapters 2-3. Addison-Wesley, Boston, MA

Copeland, B. and Sylvan, R. (1999) Beyond the Universal Turing Machine. Australian Journal of Philosophy, 77, 46-66.

Cray, S. (1996, May 30) An Imaginary Tour of a Biological Computer: Why Computer Professionals and Molecular Biologists Should Start Cooperating. Remarks of Seymour Cray to the Shannon Center of Advanced Studies, University of Virginia. http://americanhistory.si.edu/csr/comphist/montic/cray.htm (10/24/2002)

Forbes, N. (2000, November). Biologically Inspired Computing, Computing in Science and Engineering. In: http://www.computer.org/cise/articles/inspired.htm

Havran, et. al., "Introduction to Independent Component Analysis," C. Havran, L. Hupet, J. Czyz, J. Lee, L.

Horn, B. (1986). Robot vision. Cambridge, MA: MIT press, pp. 46-89.

Jesorsky, et. al. "Robust Face Detection Using Hausdorff Distance," Oliver Jesorsky, Klaus J.Kirchberg BioID AG, Berlin, Germany 2001

Kahol, "Face Recognition Using Mahalanobis Distance," Kanav Kahol. CSE 591 class presentation, Arizona.State University.

Kintsch, W., Comprehension: A Paradigm for Cognition. Cambridge, UK. Cambridge University Press, 1998

Koza, J. R.. (1994). Evolution of emergent cooperative behavior using genetic programming. In Paton, R. (Ed.), Computing With Biological Metaphors (pp. 280-297). London: Chapman & Hall.

Kung (1991), Introduction to Systolic Arrays, Addison-Wesley, 1991, Boston.

Leeuw, K., et. al. (1956). Computability by probabilistic machines. Pages 183-212 in C.E. Shannon and J. McCarthy, ed. Automatica Studies, Princeton university press, Princeton, NJ.

Minsky, M. (1988). Society of Mind. Boston: Simon and Schuster.

Ord, T. (2002) Hypercomputation: computing more than the Turing machine. Melbourne, Australia: university of Melbourne, Dept. of Computer Science, thesis, In: http://arxiv.org/abs/math.LO/0209332

Paton, R. (Ed.). (1994). Computing With Biological Metaphors. London: Chapman & Hall.

Paun, G. (2002) (http://psystems.disco.unimib.it), as viewed in October 2002.

SETI @ Home. (2002). The Search for Extra Terrestrial Intelligence. In: http://setiathome.ssl.berkeley.edu/ . Berkeley: SET @ Home.

Shackleton, M. A. and Winter, C. S. (1998). A computational architecture based on cellular processing. In Holcombe, M. and Paton, R., [Eds.] Information Processing in Cells and Tissues, pages 261--272, New York: Plenum Press.

Silc, J., Robic, B., and Ungerer, T. Asynchrony in parallel computing: from dataflow to multithreading. Parallel and Distributed Computing Practices, Vol. 1, No. 1, March 1998.

Smith, L. (2002) Tutorial On Principal Component Analysis,". University of Otago, New Zealand.

Turk and Pentland (1991), " Eigenfacesfor recognition.,". Journal of Cognitive Neuroscience, 3(1):71--86.

Turing, A. (1939). Systems of Logic based Ordinals, Proceedings of the London mathematical society, 45: 161-228.

Vandendorpe, M. Verleysen Université catholique de Louvain, Electricity Dept.

Xiaozhou, et. al., "Hausdorff Distance for Shape Matching," Yu Xiaozhou, Maylor Leung 2004